

USENIX

DALLAS CONFERENCE PROCEEDINGS

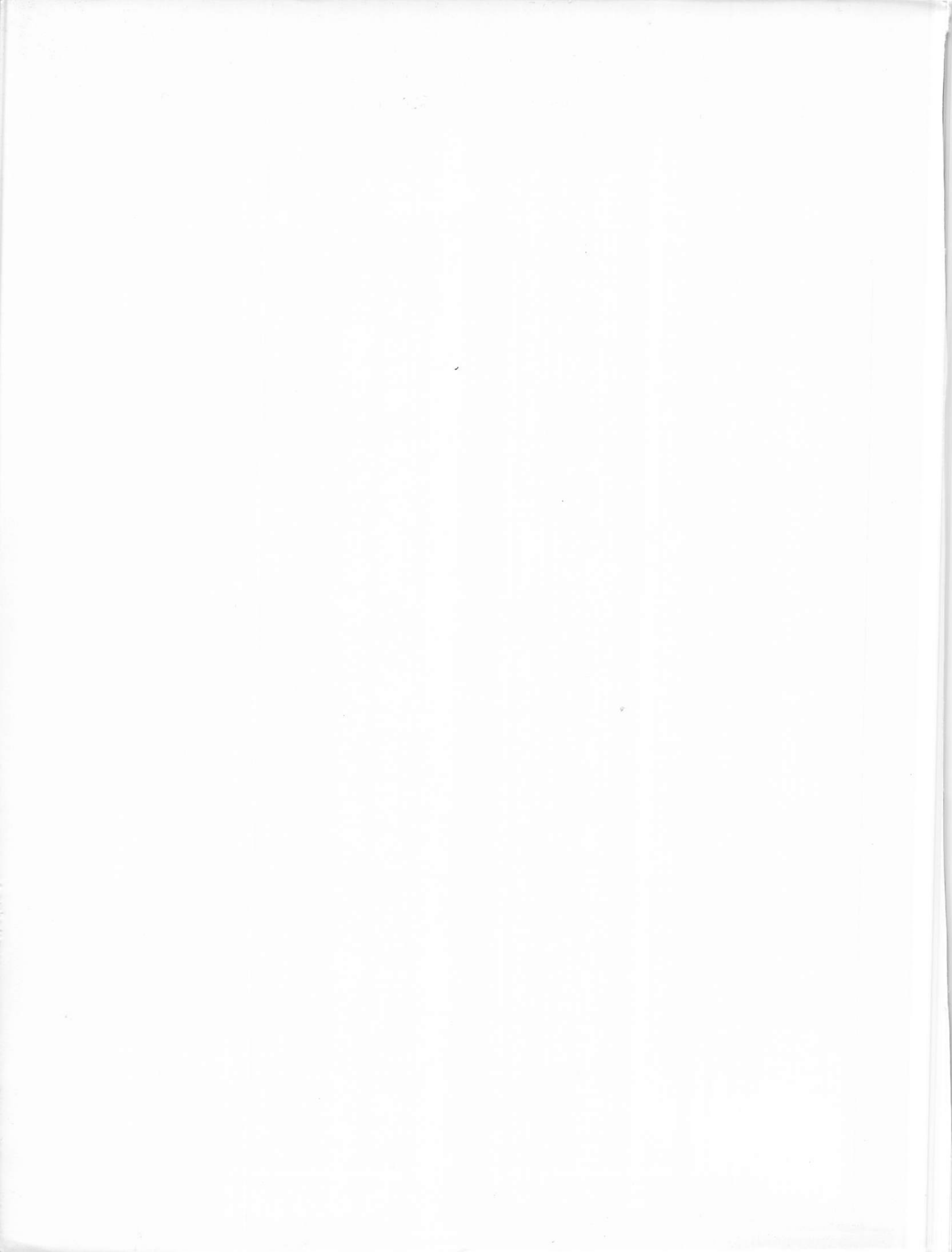
USENIX

**CONFERENCE
PROCEEDINGS**

January 21-25, 1991
Dallas, Texas

WINTER

1991



USENIX Association

Proceedings of the Winter 1991 USENIX Conference

**January 21 – January 25, 1991
Dallas, Texas, USA**

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is \$28 for members and \$34 for nonmembers.

Outside the U.S.A and Canada, please add
\$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1990 Summer Anaheim	1986 Summer Atlanta
1990 Winter Washington, DC	1986 Winter Denver
1989 Summer Baltimore	1985 Summer Portland
1989 Winter San Diego	1985 Winter Dallas
1988 Summer San Francisco	1984 Summer Salt Lake City
1988 Winter Dallas	1984 Winter Washington, DC
1987 Summer Phoenix	1983 Summer Toronto
1987 Winter Washington, DC	1983 Winter San Diego

Copyright 1991 by The USENIX Association

All rights reserved.

This volume is published as a collective work.

Rights to individual papers remain with the author or the author's employer.

AIX is a registered trademark of International Business Machines Corp.

Concentrix is a trademark of Alliant Computer Systems.

ConvexOS is a trademark of Convex Computer Corp.

CHORUS is a registered trademark of Chorus systèmes.

DECstation is a trademark of Digital Equipment Corporation.

Domain/OS is a trademark of Hewlett-Packard.

i386 is a trademark of Intel Corp.

i486 is a trademark of Intel Corp.

i860 is a trademark of Intel Corp.

IBM is a trademark of International Business Machines Corporation.

InfiniteStorage is a trademark of Epoch Systems, Inc.

MicroVAXII is a trademark of Digital Equipment Corporation.

Multimax is a trademark of Encore Computer Corporation.

MVS is a trademark of International Business Machines Corp.

Network File System is a trademark of Sun Microsystems, Inc.

NFS is a trademark of Sun Microsystems, Inc.

Nhfsstone is a trademark of Legato Systems Inc.

OSF/1 is a trademark of the Open Software Foundation.

OSx is a registered trademark of Pyramid Technology Corp.

POSIX is a trademark of the Institute of Electrical and Electronics Engineers, Inc.

PS/2 is a registered trademark of International Business Machines Corp.

REAL/IX is a trademark of Modular Computer Systems, Inc.

Renaissance is a trademark of Epoch Systems, Inc.

RISC System/6000 is a trademark of International Business Machines Corporation.

RSTS is a trademark of Digital Equipment Corp.

RTU is a trademark of Concurrent Computer Corp.

SunOS is a trademark of Sun Microsystems, Inc.

System/370 is a trademark of International Business Machines Corp.

System/390 is a trademark of International Business Machines Corp.

Ultrix is a trademark of Digital Equipment Corporation.

UNICOS is a registered trademark of Cray Research, Inc.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

UTS is a trademark of Amdahl Corp.

VAX is a trademark of Digital Equipment Corporation.

VMS is a trademark of Digital Equipment Corp.

WE is a registered trademark of AT&T.

TABLE OF CONTENTS

Preface	vii
Program Committee	viii
Author Index	ix

PLENARY SESSION

Wednesday (9:00-10:30)

Chair: Lori Grob

- Opening Remarks and Announcements
Lori S. Grob, Chorus systèmes
- Keynote Address: Graphics as Systems Programming
Eben Ostby, Pixar

Kernels I

Wednesday (11:00-12:30)

Chair: Barry Gleeson

Processors, Priority, and Policy: Mach Scheduling for New Environments	1
<i>David L. Black, Carnegie Mellon University</i>	
A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility	13
<i>Marc Guillemont, Jim Lipkis, Doug Orr, Marc Rozier, Chorus systèmes</i>	
Partitioned Multiprocessors and The Coexistence of Heterogeneous Operating Systems	23
<i>Nick Vasilatos, Concurrent Computer Corporation</i>	

File System Performance

Wednesday (2:00-3:30)

Chair: Trent Hein

Extent-like Performance from a UNIX File System	33
<i>L. W. McVoy, S. R. Kleiman, Sun Microsystems, Inc.</i>	
Smart Filesystems	45
<i>Carl Staelin, Hector Garcia-Molina, Princeton University</i>	
Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol	53
<i>Rick Macklem, University of Guelph</i>	

Threads & Networks

Wednesday (4:00-5:30)

Chair: Deborah Scherrer

SunOS Multi-thread Architecture	65
<i>M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, Sun Microsystems Inc.</i>	
Bringing the C Libraries With Us into a Multi-Threaded Future	81
<i>Michael B. Jones, Carnegie Mellon University</i>	
A Tree-Based Packet Routing Table for Berkeley Unix	93
<i>Keith Sklower, University of California, Berkeley</i>	

Interface Tools

Thursday (9:00-10:30)

Chair: Tom Duff

An X11 Toolkit Based on the Tcl Language	105
<i>John K. Ousterhout, University of California at Berkeley</i>	
User Interface Construction Based On Parallel and Sequential Execution Specification	117
<i>Toshiyuki Masui, Center for Machine Translation, Carnegie Mellon University</i>	
SHOME MOVIE - Tools for Building Demos on a Sparcstation	127
<i>Stephen A. Uhler, Bellcore</i>	

Awk Paper & Kernel Panel

Thursday (11:00-12:30)

Chair: Marc Donner

Awk As A Major Systems Programming Language	137
<i>Henry Spencer, University of Toronto</i>	
Panel: Kernel Directions	
<i>Michel Gien, Chorus systèmes; Michael Karels, University of California, Berkeley; Mike Powell, Sun Microsystems; Rick Rashid, Carnegie Mellon University</i>	
<i>Marc Donner (Moderator), IBM Research</i>	

Programming Tools

Thursday (2:00-3:30)

Chair: Marc Donner

Program Loading in OSF/1	145
<i>Larry W. Allen, Harminder G. Singh, Kevin G. Wallace, Melanie B. Weaver, Open Software Foundation</i>	
Compiling from Saved State: Fast Incremental Compilation with Traditional UNIX Compilers	161
<i>Alastair Fyfe, Ivan Soleimanipour, Vijay Tatkar, Sun Microsystems</i>	
A New Hashing Package for UNIX	173
<i>Margo Seltzer, University of California, Berkeley; Ozan Yigit, York University</i>	

File Systems

Thursday (4:00-5:30)

Chair: Steve Bourne

- Evolutionary Path to Network Storage Management 185
Robert K. Israel, Antony W. Foster, Arun Taylor, Tracy M. Taylor, Neil Webber, Epoch Systems, Inc.
- A Highly Available Network File Server 199
Anupam Bhide, IBM T. J. Watson Research Center; Elmootazbellah N. Elnozahy, Rice University; Stephen P. Morgan, IBM T. J. Watson Research Center
- The OSF/1 UNIX Filesystem (UFS) 207
Susan LoVerso, Noemi Paciorek, Alan Langerman, Encore Computer Corporation; George Feinberg, Open Software Foundation

Objects in Action

Friday (9:00-10:30)

Chair: Michel Gien

- Advancing Files to Attributed Software Objects 219
Andreas Lampen, Technische Universität Berlin
- Organizing Tools in a Uniform Environment Framework 231
Axel Mahler, Technische Universität Berlin
- The Process File System and Process Model in UNIX System V 243
Roger Faulkner, Sun Microsystems; Ron Gomes, AT&T Bell Laboratories

Insecurity

Friday (11:00-12:30)

Chair: Michael Karels

- Limitations of the Kerberos Authentication System 253
Steven M. Bellovin, Michael Merritt, AT&T Bell Laboratories
- UNIX Password Encryption Considered Insecure 269
Philip Leong, University of Sydney; Chris Tham, State Bank of Victoria
- An Authentication Mechanism for USENET 281
Matt Bishop, Dartmouth College

Distributed File Systems Panel

Friday (11:00-12:00)

Chair: Peter Honeyman

- Distributed File Systems Panel
*Rafael Alonso, Princeton University; Mike Kazar, Transarc; John Ousterhout, University of California, Berkeley; Brian Palowski, Sun Microsystems
Peter Honeyman (Moderator), IFS, University of Michigan*

Kernels II

Friday (2:00-3:30)

Chair: Jan Edler

An Experimental Implementation of Draft POSIX Asynchronous I/O	289
<i>A. Lester Buck, Robert A. Coyne, Jr., IBM Federal Sector Division, Houston</i>	
The Parallelization of UNIX System V Release 4.0	307
<i>Mark Campbell, Richard Barton, Jim Browning, Dennis Cervenka, Ben Curry, Todd Davis, Tracy Edmonds, Russ Holt, John Slice, Tucker Smith, Rich Wescott, NCR Corporation—E&M Columbia</i>	
A NonStop UNIX Operating System	325
<i>Peter Norwood, Tivoli Systems, Inc.</i>	

Distributed Processing

Friday (4:00-5:30)

Chair: Max Meredith Vasilatos

DRUMS: A Distributed Statistical Server for STARS	335
<i>Andy Bond, John H. Hine, Victoria University of Wellington</i>	
Experience Building a Process Migration Subsystem for UNIX	349
<i>Dan Freedman, University of Calgary</i>	
A Modular Architecture for Distributed Transaction Processing	357
<i>Michael Wayne Young, Dean S. Thompson, Elliot Jaffe, Transarc Corporation</i>	

PREFACE

Over the last few years I have noticed a substantial change in the field of operating systems development. I have found that ambitious systems that were formerly confined to universities and research labs had escaped, if you like, and were turning up in offices and computer centers everywhere.

These changes have affected my own life as well. I have found myself leaving the university research environment to work in Europe for a commercial concern. USENIX also has changed over the last few years. It has gone from a small organization of people doing UNIX development and system programming to an organization of thousands of people from all over the world working on all aspects of use, development and support of computers and workstations.

As an American living and working in Europe in 1989 and 1990, I have seen enormous changes and these changes have had an impact on my thinking. When the time came to consider what kind of conference I wanted this to be, I decided that I wanted it to reflect the sort of growth and change that I have been speaking about.

The unofficial theme of this conference asks, "What's next?" We wanted to look beyond the current standards battles and provide some insight to the questions of, "What kind of systems will I be using in the next 20 years?", "Will it be UNIX as I know it, a direct descendent, or a distant relation?", "What kinds of applications will I be running?"

In order to try and answer these questions we reached out and sought submissions on "futuristic" operating systems and novel applications areas.

The response to the call for papers was impressive. We chose the papers to be presented from among 84 submissions – which came from 14 different countries.

In order to further answer or perhaps further confuse the question of "What's next?", we are presenting two panel sessions.

One panel session will debate and discuss the transitions that are taking place or appear to be taking place from UNIX systems with large all-inclusive kernels to micro kernels. For this panel we were lucky enough to persuade such experts as Michel Gien, Michael Karels, Michael Powell, and Richard Rashid to appear on the panel. We were equally lucky to have Marc Donner to moderate it.

The other panel session grew out of a discussion which started at the previous USENIX conference, in Summer 1990. This panel will discuss the future of distributed file systems. We are happy to have Rafael Alonso, Micheal Kazer, John Ousterhaut and Brian Pawlowski on the panel and again equally happy to have Peter Honeyman to referee.

This conference also sees the further growth of the Concurrent Sessions – now called the Invited Talks.

We are happy to have Eben Ostby of Pixar to give the keynote speech. As the majority of attendees do not get to attend graphics or animation conferences we hoped that this might give them some exposure to what is surely one of the most interesting uses of computers today.

There are too many people who have contributed their time and effort to this conference to thank them all by name, but I would like to thank especially the following people: the Program Committee who gave above and beyond the call of duty: Steve Bourne, Marc Donner, Tom Duff, Jan Edler, Barry Gleeson, Michel Gien, Trent Hein, Andrew Hume, Michael J. Karels, Deborah K. Scherrer, Melinda Shore, Max Meredith Vasilatos; Judy Desharnais and Ellie Young who gave advice and encouragement, Trent Hein who made sure that it would be possible to have a proceedings, Rob Kolstad for processing the pictures and macros, and the University of Colorado at Boulder Crew (Brian Drake, Darren Hardy, Andy Kuo, Herb Morreale, and liaison Evi Nemeth) who actually made it happen.

Lori S. Grob
Program Chair

TECHNICAL PROGRAM COMMITTEE

Steve Bourne
Sun Microsystems



Andrew Hume
AT&T Bell Labs



Marc Donner
IBM Research



Michael J. Karels
UC Berkeley



Tom Duff
AT&T Bell Labs



Deborah K. Scherrer
mt Xinu



Jan Edler
NYU Ultracomputer Research Lab



Melinda Shore
mt Xinu



Barry J. Gleeson
Unisys Corporation



Michel Gien
Chorus systèmes



Lori Grob, *Chair*
Chorus systèmes



Max Meredith Vasilatos



Trent Hein
University of Colorado, Boulder

CONFERENCE ORGANIZERS

Lori Grob, *Technical Program Chair*
(Chorus systèmes)
Judith F. Desharnais, *Meeting Planner*
(USENIX Association)
Dan Klein, *Tutorial Coordinator*
(USENIX Association)
Sonya Neuffer, *Conference Computer Communications*
(Canstar, Inc.)
Eve Podet, *Conference Computer Communications*
(mt Xinu)

PROCEEDINGS PRODUCTION

Trent Hein, *Proceedings Coordination*
(University of Colorado, Boulder)
Rob Kolstad, *Pictures, Typesetting, & Macros*
(Sun Microsystems)
Brian Drake, Darren Hardy, Andy Kuo,
Herb Morreale, *Typesetting*
(University of Colorado, Boulder)
Evi Nemeth, *Liaison*
(University of Colorado, Boulder)

AUTHOR INDEX

Larry W. Allen	145	Rick Macklem	53
Richard Barton	307	Axel Mahler	231
S. Barton	65	Toshiyuki Masui	117
Steven M. Bellovin	253	L. W. McVoy	33
Anupam Bhide	199	Michael Merritt	253
Matt Bishop	281	Stephen P. Morgan	199
David L. Black	1	Peter Norwood	325
Andy Bond	335	Doug Orr	13
Jim Browning	307	John K. Ousterhout	105
A. Lester Buck	289	Noemi Paciorek	207
Mark Campbell	307	M.L. Powell	65
Dennis Cervenka	307	Marc Rozier	13
Robert A. Coyne	289	Margo Seltzer	173
Ben Curry	307	D. Shah	65
Todd Davis	307	Harminder G. Singh	145
Tracy Edmonds	307	Keith Sklower	93
Elmootazbellah N. Elnozahy	199	John Slice	307
Roger Faulkner	243	Tucker Smith	307
George Feinberg	207	Ivan Soleimanipour	161
Antony W. Foster	185	Henry Spencer	137
Dan Freedman	349	Carl Staelin	45
Alastair Fyfe	161	D. Stein	65
Hector Garcia-Molina	45	Vijay Tatkar	161
Ron Gomes	243	Arun Taylor	185
Marc Guillemont	13	Tracy M. Taylor	185
John H. Hine	335	Chris Tham	269
Russ Holt	307	Dean S. Thompson	357
Robert K. Israel	185	Stephen A. Uhler	127
Elliot Jaffe	357	Nick Vasilatos	23
Michael B. Jones	81	Kevin G. Wallace	145
S. R. Kleiman	33	Melanie B. Weaver	145
S. R. Kleiman	65	Neil Webber	185
Andreas Lampen	219	M. Weeks	65
Alan Langerman	207	Rich Wescott	307
Philip Leong	269	Ozan Yigit	173
Jim Lipkis	13	Michael Wayne Young	357
Susan LoVerso	207		

Processors, Priority, and Policy: Mach Scheduling for New Environments

David L. Black¹ - Carnegie Mellon University

ABSTRACT

Changing hardware and software environments require alternatives to the timesharing scheduling policies supported by Unix, Mach, and similar systems. Effective use of multiprocessor and multicomputer architectures often requires dedicating processors to some applications. Complex real-time applications demand the level of services available in a Unix-like environment, but such applications cannot be timeshared. These and other new environments require alternatives to the traditional timesharing scheduling model.

This paper describes scheduling techniques that enable the Mach operating system to support new application environments. Mach's processor allocation facility supports dedicating processors to applications. Removing allocation decisions from the kernel and implementing them in a separate server allows a single kernel to support a wide variety of allocation policies and application environments. The Mach system also supports scheduling policy alternatives to timesharing. Fixed priority scheduling is currently implemented for use in real time environments, and the design of the kernel interface permits additional policies to be added. These facilities are designed to work together, giving an application complete control over scheduling of processors dedicated to it. Appendices to this paper describe the interfaces to both the kernel and a simple gang scheduling server.

Introduction

Unix and related systems are being used in new environments that are ill suited to traditional timesharing scheduling policies. Parallel and concurrent programming techniques change the nature of the scheduling problem by splitting applications into cooperating independently scheduled entities (e.g., processes, threads). This cooperation violates the traditional timesharing assumption that all processes are in competition for the resources of the machine, making the goal of timesharing (equal division of processing resources) potentially inappropriate for such applications. A similar situation occurs in the area of real time computing, where factors other than accumulated usage are much more important in determining scheduling order. In such environments, the most time-critical portions of applications may need to consume disproportionate shares of the available processor time. Both of these are examples of new environments that cannot be adequately supported by timesharing. Alternative scheduling techniques are required to support the use of Unix and similar systems in these environments.

This paper describes the Mach scheduling facilities that support explicit resource allocation for these environments. Mach's processor allocation facility allows processors to be dedicated to specific uses or applications, and permits applications to exercise explicit control over how these processors are used. This has been used for gang scheduling of multiprocessor applications and to support user-mode processor scheduling. The scheduling policy interface allows the use of non-timesharing scheduling policies for different environments. A fixed priority scheduling policy has been implemented for real time and related environments. The policy interface is extensible to allow other policies to be added.

Mach Background

Mach [8] is a portable multiprocessor operating system developed at Carnegie Mellon University. It has been ported to and used on a variety of multiprocessor platforms, including multiprocessor VAXes (784, 6000 series, and 8000 series models), the Encore Multimax, and the Sequent Symmetry. Mach is the basis for the multiprocessor support in the Open Software Foundation's OSF/1 operating system. The Mach system is based on a small number of fundamental abstractions implemented by a communication oriented kernel. Most kernel operations are invoked by sending messages to the kernel, permitting transparent remote invocation over networks.

The Mach kernel exports five basic abstractions; the task, thread, port, message, and memory object, collectively referred to as *objects*. A task is

¹This Research was supported by the Defense Advanced Research Projects Agency (DOD) and monitored by the Space and Naval Warfare Systems Command under Contract N00039-87-C-0251, ARPA Order No. 5993. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. government.

an execution environment in which threads may run, and is also the basic unit of resource allocation, consisting of a paged virtual address space and access to resources (via ports). A thread is a locus of control within a task. A port is a capability-protected communication channel with exactly one receiver and one or more senders. A message is a typed collection of data elements; communication is performed by sending messages to ports. A memory object is a region of data provided by a server that can be mapped into a task. Memory objects can be used to implement functionality such as network shared memory and mapped files.

Ports are used to export Mach kernel objects to applications. Each object is represented by a port, and object operations are invoked by sending messages to the corresponding ports. Results (if any) are returned to the sender in a second message; this pair of messages constitutes a remote procedure call (RPC) to the kernel. The capability semantics of ports are used to protect all objects implemented by the Mach kernel. Port access is controlled by kernel-managed capabilities, or *rights* to send or receive messages on a port. The initial creator of a port receives both send and receive rights to the port; thereafter, rights may only be transferred by Mach IPC messages. This provides a strong capability-based model of protection; an application can access a kernel object only if the application has specifically obtained a send right for the corresponding port. Even if the application can learn the identity of some port via other means, the kernel will not permit the application to use that port without a send right.

Processor Allocation

Mach's processor allocation facility supports dedicating various processors of a multiprocessor to different uses on a short or long term basis. Responsibility for this allocation is divided among three components; the kernel, a privileged scheduling server, and the applications themselves. The kernel implements processor allocation mechanisms, with policy decisions being made outside the kernel by a privileged scheduling server and the applications. These components and their relationships are shown in Figure 1. There is a single privileged scheduling server per system that is responsible for allocating processors to competing applications, with the applications being responsible for intra-application allocation policy. The kernel interface is specified by the facility, but the application to server interface is not; this allows the latter interface to change in order to support different allocation policies. Mach's processor allocation facility can be used to implement gang scheduling of applications and application-specific schedulers, among other uses.

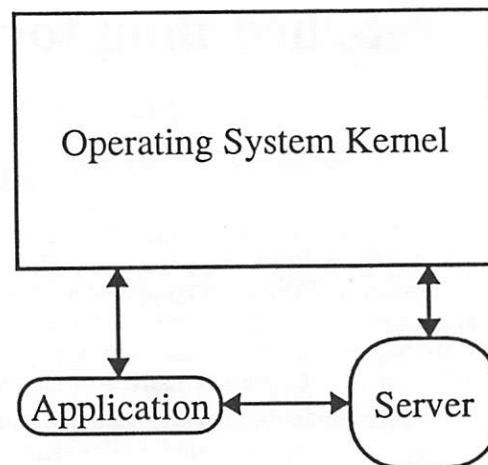


Figure 1: Processor Allocation Components

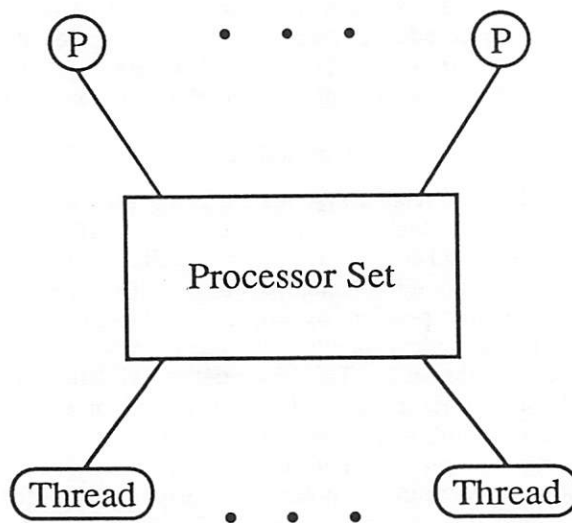


Figure 2: Mach Processor Set

The kernel interface for processor allocation introduces three new entities to those exported by the Mach kernel:

processor_set - This is a set of processors on which threads can execute. It is an independent object to which both threads and processors can be assigned, as shown in Figure 2. There is a distinguished set, the default set, to which all threads and processors are initially assigned. Two ports are exported by the kernel for a processor set, a name port for obtaining information about the set, and a control port for performing operations on it.

host - This represents the host, a computer running a single Mach kernel. There are two versions of this object: a non-privileged version for information queries, and a privileged version that grants the

right to manipulate physical resources. Other resource operations (e.g., making memory non-pageable) will be added in the future. The non-privileged port also serves as a name port for the host.

processor - This corresponds to a hardware processor. One of the processors is distinguished as the master processor; execution of unparallelized kernel code is restricted to this processor.

The processor set concept is introduced to achieve the flexibility required to support different programming models. A key characteristic of programming models for parallel applications is whether the number of kernel level virtual processors (threads, processes, etc.) exceeds the number of physical processors on which the application is intended to be run. Binding threads to individual processors is inadequate when this is the case. Such models require binding a pool of threads to a pool of processors; the notion of a processor set is introduced to make these pools explicit. The processor set serves as the target for assignment operations that add both threads and processors to the respective pools. The privileged server is responsible for processor assignment, with applications being responsible for thread assignment. This frees the server from dependence on the internal structure of complex applications, and allows applications to do their own scheduling via control over thread assignments.

The host concept is introduced to isolate authentication concerns from the processor allocation interface. Host and processor operations are privileged because the required ports are only available to privileged servers and applications. An exception is the unprivileged operations that obtains information about hosts. The kernel provides allocation mechanisms for processors; policy is the responsibility of the server. In addition, the servers may understand more about the topology of the machine (e.g., clustering of processors) than the kernel. Processor sets are not privileged, and are intended to form a basis for the interfaces exported by the privileged servers. Applications cannot obtain ports for the processors assigned to their processor sets.

Each processor, task, and thread is always assigned to exactly one processor set. A pool of processors is assembled by assigning processors to a processor set; a processor assigned to a set will only run threads that have been assigned to that set. The master processor must always be assigned to the default set² This is necessary to ensure that internal kernel threads and important daemons have a processor on which they can execute. A processor only executes threads that are assigned to its processor

set, and threads only execute on processors assigned to their processor set.³ Task assignments are used only for the purposes of determining the initial assignment of newly created tasks and threads; tasks inherit their initial assignment from their parent, and threads inherit their initial assignment from the task that contains them. These assignments may be subsequently changed.

The following steps illustrate how an application could allocate six processors for its use:

1. **Application** → **Kernel** : Create processor set.
2. **Application** → **Server** : Request six processors for processor set.
3. **Application** → **Kernel** : Assign threads to processor set.
4. **Server** → **Kernel** : Assign processors to processor set.
5. **Application** : Use processors.
6. **Application** → **Server** : Finished with processors (Optional).
7. **Server** → **Kernel** : Reassign processors.

This example illustrates three important features of the allocation facility. The first is that the application creates the processor set and uses it as the basis of its communication with the server, freeing the server from dependence on the internal structure of the application. The second is that only one processor set is used. The standard Mach timesharing scheduler is used within each processor set; for this example, an important feature is that if the task contains six or fewer threads there will be no context switches to shuffle the threads among the allocated processors. The third feature is that the server does not need the application's cooperation to remove processors from it. The server retains complete control over the processors at all times because it retains the access rights to the processor objects. Removing processors without the application's cooperation should not be necessary for well-behaved applications, but can be useful for removing processors from a runaway application that has exceeded its allotted time.

Implementation

The kernel implementation of processor sets is an extension of the Mach timesharing scheduler. The same scheduling algorithms are used within each processor set to avoid dependencies on dedicated processors; applications may still behave differently when assigned to dedicated processors, but the cause of this behavior is to be found in the

²In systems that do not have a master processor, this invariant can be replaced by: the default set must always be assigned at least one processor.

³Unparallelized kernel code causes exceptions to this rule; threads may be temporarily forced to the master processor to execute it.

applications, not the scheduler. The data structure for each processor set contains a run queue for threads. A list of idle processors is maintained on a per processor set basis because a processor can only be dispatched to threads that are assigned to its current processor set. The processor set data structure is also the head of individual lists that are linked through the data structures of assigned tasks, threads, and processors so that these entities can be found and reassigned when the processor set is terminated. In addition, the data structure contains some state information required to run the timesharing scheduling algorithm (see [2] for details), the identities of the ports that represent the set, and a mutual exclusion lock to control access to the data structure. Redirection of device interrupts away from processors not assigned to the default set is machine dependent; current implementations do not include this redirection.

Processor sets with no processors, or *empty* processor sets, are an important feature of this design. The use of empty processor sets is important to isolating application structure from the server. Empty processor sets also allow the server to do coarse timeslicing of processors among several applications; the applications without processors are in empty processor sets waiting their turn to run. Threads being assigned to empty processor sets are first suspended to ensure that the assignment does not occur while the thread is waiting for an important event (e.g., completion of a disk access), as the kernel prevents suspension of threads waiting for such events. Threads are actually suspended for the duration of all assignment operations, and left suspended if the target processor set is empty. An obvious exception is the case of a thread changing its own assignment; the thread is suspended when the assignment is complete if the target processor set is empty. This suspension logic also implements the required change of processors when the target processor set is not empty; a thread assigning itself substitutes an explicit context switch. Processor assignment operations suspend all the threads in a processor set when removing the last processor, and resume them when adding the first processor.

Special techniques are used to manage processor to processor set assignments. Code in a critical path of the scheduler reads the assignment as part of finding a thread to run on a processor. To optimize this common case against infrequent changes in assignment, each processor is restricted to changing only its own assignment. This eliminates the need for a mutual exclusion lock because a processor looking for a new thread cannot simultaneously change its assignment. The cost to the assignment operation is that it must temporarily bind a thread to the processor while changing the assignment. An internal kernel thread, the action thread, is used for this purpose. Current kernels use only one action

thread, but are designed to accommodate more. The processor assignment interface allows a server to avoid synchronizing with completion of each assignment to exercise the parallelism available from multiple action threads. Interprocessor interrupts are used for changing the assignment of processors and threads as needed.

Table 1 reports the times required by some basic operations in the processor allocation system on an Encore Multimax with ns32332 processors (approximately 2 MIPS). The times are shown as mean \pm standard deviation in microseconds. The self and other cases of thread assignment correspond to a thread assigning itself and a thread assigning another thread. These times are easily amortized by the expected assignment durations of multiple seconds to multiple minutes. Further optimization including the use of server threads as action threads can improve these times when support of shorter assignment durations becomes important.

Operation	Time
Create Processor Set	2250
Assign Processor	4772
Assign thread (self)	1558
Assign thread (other)	2624

Table 1: Multimax Allocation
Operation Performance

A Gang Scheduling Server

A simple processor allocation server for gang scheduling has been implemented as part of this work. This server is a batch scheduler for processors with limits of 75 of the total processors on the machine and 15 minutes maximum request length. These limits are based on the usage environment at CMU, and are examples of policy parameters that will vary from site to site. The server satisfies requests in a greedy fashion with strict adherence to the order in which they are received. For example, if the server has 10 processors to allocate and receives requests for 4, 7, and 2 processors, it will satisfy the request for 4 first, and then the requests for 7 and 2 together. The request for 2 processors will not be moved ahead of the request for 7 even though processors are available for the former request. This algorithm was chosen for its simplicity and lack of starvation; more sophisticated algorithms that make better use of the processors by satisfying requests out of order could be used.

The server exports an object-oriented remote procedure call interface to applications. The interface is based on request objects that consist of two components:

- A time duration for the allocation request.
- A sequence of <processor set, number of processors> pairs.

Request objects are represented to the clients of the server by individual Mach ports (like kernel objects). A request is satisfied by assigning each processor set its corresponding number of processors for the time duration specified. Allocations may be terminated before the end of their time durations. An application interacts with the server by creating a request object, adding <processor set, number of processors> request pairs to it, and activating the request. A separate interface routine destroys the request and releases any associated processors for reallocation by the server. Additional interface routines provide information about the server and individual requests. The interface also supports several optional services implemented by the server. The *destroy* option destroys the processor sets associated with a request when the allocation expires or is terminated. This is useful for novices because it prevents their programs from being suspended when an allocation is exceeded; the programs continue to run, but in the default processor set. The *notify* option causes the server to send messages to the application after allocating processors and one second before deallocating them. Since allocation and deallocation of multiple processors is not atomic, an application can use these messages to ensure that it is never running with less than its full complement of processors. The *repeat* option causes the request to repeat, allowing the server to timeslice the machine among applications that need more than 15 minutes of time (using 15 minute timeslices). Finally the *task* option tells the server that the application is a task with one or more threads; this allows the server to optimize removing processors from the application by suspending it first.

The server implementation uses multiple threads, shared memory, and message passing. One thread manages communication with the application clients, and a second thread manages the actual allocation and deallocation of the processors. The primary interaction between these threads is via operations on shared data structures describing the requests, but the interaction thread sends a message to the processor thread when an immediate change to the assignment of processors is needed. One such situation is the activation of an allocation request that can be immediately satisfied. The Mach Interface Generator (MiG) is used to generate remote procedure call stubs for the server interfaces, freeing applications from the details involved in marshalling arguments and formatting messages. On the server side, MiG-generated stubs transparently invoke routines that translate from the external representation of request objects (ports) to the corresponding internal data structures.

Library routines have been implemented to hide the server interfaces, so that an application can make a single call indicating how many processors it wants for how many seconds. This routine contacts

the server, arranges the allocation, and returns when the server has begun to assign the requested processors. Additional routines incorporate some of the other options available from the server, such as the *notify* option. The total time taken by this routine is about 35ms to allocate one processor plus the processor assignment time of about 5ms per additional processor. This overhead is acceptable given the expected allocation durations of tens of seconds to tens of minutes. Both the basic server interface and the library routines that hide it are documented in appendix B.

The *cpu_server* and library interfaces have been successfully used by researchers and students in an undergraduate parallel programming course at Carnegie Mellon to facilitate performance measurements of parallel programs. The server removed almost all of the administrative difficulties usually involved in obtaining dedicated machine time, by allowing users to obtain dedicated processors at any time on short notice. In addition, development of the server repeatedly demonstrated the utility of implementing policy in a separate server because server crashes did not crash the operating system.

Many extensions and changes to the policy implemented in the *cpu_server* are possible. Since it is a batch scheduler for processors, techniques originally developed for batch scheduling of memory, such as assigning higher priority to shorter requests, are applicable. In addition, the server could be extended to allow some users higher or absolute priority in allocating processors, or to allow more processors to be allocated during light usage periods. Finally, the server can be replaced in its entirety by a server that implements a different scheduling policy. One promising new policy is to vary the number of processors available to applications based on the overall demand for processors. A server with this policy can notify applications to reconfigure when it changes the number of processors available. Researchers at Stanford are pursuing this approach and have implemented a server for this scheduling policy under Mach with good initial results [9]. The major benefit of using Mach's processor allocation facility for this policy is that it protects cooperative applications (that reduce their demand for processors when the load on the system rises), from uncooperative applications (that do not). The strict division of processors into processor sets forces the uncooperative applications to compete against themselves instead of other (cooperative) applications. Most of these extensions require changes to the interface between applications and the server (e.g., to transmit user authentication information). This illustrates the flexibility of not specifying the application to server interface as part of the operating system kernel.

Scalability

The scalability of Mach's processor allocation facility is an important issue because it is intended to adapt to future (larger) multiprocessor architectures. Scalability is of interest for both the overall system, and its individual pieces (kernel interface, kernel implementation, `cpu_server` and other servers). The architecture of the multiprocessor being managed is a key factor; if the architecture is not scalable, then software that depends on it will also not be scalable. The design of Mach's processor allocation facility is scalable because it is independent of the multiprocessor architecture being managed. Portions of the current implementation are not scalable to the extent that they assume a non-scalable multiprocessor architecture.

An important scalability distinction exists between uniform memory access (UMA) and non-uniform memory access (NUMA) multiprocessor architectures. All memory in UMA architectures is equidistant (in access time) from all processors. This limits scalability because access time rises with the number of processors. All current realizations of UMA architectures have scalability limitations caused by bus bandwidth and/or available ports on multiported memory modules. NUMA architectures allow different access times to memory from different processors; by placing some memory close to each processor, they can take advantage of locality to keep the average access time down while the worst case time increases. As a result, most NUMA architectures are scalable. When large caches are employed in UMA architectures to mask longer memory access times, UMA architectures may behave like NUMA architectures from a scheduling standpoint. This occurs when the size of an application's cached state (footprint) and its longevity become more important to scheduling than load balancing and utilization of idle processors. The resulting scheduling policy must be more concerned with running thread's whose state is in local memory (cache), a primary NUMA concern, instead of achieving the best utilization of processors, a primary UMA concern. A useful criterion is that if it ever useful to stall a runnable thread (because no preferred processor is available) while some other processor is idle, then the machine is a NUMA for scheduling purposes. The conclusion for processor allocation facilities, is that scalability requires the ability to support and manage NUMA multiprocessor architectures.

The overall design of Mach's processor allocation facility is scalable, but portions of the current implementation that depend on a UMA multiprocessor architecture are not. The kernel interface is scalable, as it is object oriented and can support parallel operations on independent objects. The current kernel implementation is not scalable, but can be made scalable with some small changes. In addition to

adding more action threads to the kernel, data structures may need to be changed so that different threads manage different pools of processors (this allows the threads' kernel stacks to be bound to specific cluster memories in a NUMA architecture); these changes are easy to make. Outside the kernel, the current `cpu_server` is designed for and assumes a UMA architecture. This server is not scalable, in part because the allocation policy it implements (all processors are treated as one large pool) is only appropriate for UMA architectures. Applications for NUMA architectures will likely want multiple smaller pools of processors to match the architecture, and hence use multiple processor sets in their interaction with the scheduling server. An alternative for NUMA architectures is to implement the processor allocation service as a collection of cooperating servers to decentralize the resource management and obtain a structure in which servers run in proximity (measured by memory access delays) to the processors they manage. A promising structure for such a collection of servers is presented by Feitelson and Rudolph [3], although their proposal to use dedicated processors for the servers is questionable. A software implementation of their server hierarchy and algorithms would be a better match to the small overhead of current and proposed processor allocation techniques.

Heterogeneity

A heterogeneous multiprocessor is a shared memory multiprocessor whose processors exhibit some important incompatibility (usually different instruction sets) among themselves. These machines can be divided into the classes of *incompatible heterogeneity* and *compatible heterogeneity*. Machines exhibiting incompatible heterogeneity have more than one class of processors, but each application can only be executed on processors from a single class (but this class may differ from application to application). A multiprocessor consisting of i386 and i860 processors is an example of incompatible heterogeneity because these two processor classes have completely incompatible instruction sets. Machines exhibiting compatible heterogeneity have some applications that can run on more than one class of processors. A multiprocessor consisting of i386 and i486 processors is an example of compatible heterogeneity. i386 applications can be run on i486 processors, but some i486 applications cannot be run on i386 processors because the i486 contains instructions that are not found in the i386. It is possible to mix these two types of heterogeneity, for example in a multiprocessor that contains i386, i486, and i860 processors.

Mach's processor allocation facility supports incompatible heterogeneity, but not compatible heterogeneity. The processor allocation facility's ability to divide a multiprocessor into sets of

processors supports incompatible heterogeneity by ensuring that threads which can only run on a particular class of processor do only run on processors in that class. On the other hand, it does not support compatible heterogeneity because the required boundaries are not strict; some threads can cross them while others cannot. It is the author's view that compatible heterogeneity is an orthogonal issue to processor allocation, and hence the data structures that support it should be replicated for each processor set. The Alliant scheduler (discussed in the related work section) takes a different approach that integrates the data structures to support compatible heterogeneity and processor allocation.

Scheduler Priorities and Policies

This section describes the Mach system operations that allow applications to control scheduling priorities and policies for threads. These operations are a natural extension of the processor allocation interface, utilizing the idea that the control port for a processor set represents the privilege to control scheduling of any processors assigned to the set. Thus the processor sets divide a host into independent scheduling domains that are managed by different schedulers. Safeguards are incorporated in the thread assignment operation to protect processor sets when threads are reassigned from processor sets with different scheduling policies and assumptions. Overhead considerations dictate that short term scheduling decisions (e.g., when to context switch, which thread to run next) be made in the kernel. Therefore the kernel implements short term scheduling policies, and provides an interface that allows these policies to be selected on a per-thread basis.

Each thread has both a priority and a maximum priority that are used as inputs to the scheduling policy; these priorities range from 0 to 31 with lower numbers corresponding to higher priorities. Thread priorities are controlled by applications, and a maximum priority is provided to allow a (user) scheduler to limit the priorities at which threads can run. A thread's priority is never greater than the maximum priority and the maximum priority can only be decreased by the thread. The maximum priority can be reset to any value by presenting the control port for the thread's processor set. Since the default processor set's control port is privileged, applications that do not use processor allocation cannot raise their thread priorities above their initial maximum. A thread inherits its initial priority from its containing task and its initial maximum priority from its initial processor set (the one to which its task is assigned) when the thread is created. Tasks, in turn, inherit their priority from their parents on creation. A processor set's maximum priority serves to initialize any threads created in the processor set, and to defend the processor set against any threads subsequently assigned to it; if any such threads have

priorities greater than the processor set's maximum priority, these priorities are reset to that maximum priority (likewise for the threads' maximum priorities). The initial priority of the first task, and the initial maximum priority of any newly created processor set (including the default processor set at boot time) are both set to 12 to correspond to the Unix (4.3 BSD) default maximum priority value of 50 from a range of 0-127.

The Mach kernel also supports the concept of a per-thread scheduling policy. A scheduling policy is responsible for mapping from thread priorities (kernel interface) to the underlying priorities used by the kernel's context switch mechanism. Currently, only timesharing and fixed priorities are supported, but the interface is extensible to support additional policies in the future (e.g., round-robin is appropriate in some circumstances). The timesharing policy uses a multi-level usage-based feedback mechanism to produce scheduling priorities lower than the base priorities for usage balancing. The fixed priority policy maps thread priorities directly onto internal priorities. A processor set contains a mask of allowed policies, but this mask is only enforced on thread creation, assignment, or policy change. This allows a number of fixed priority threads to be created by an application that then resets this mask to prevent the creation of others. The mask must always allow timesharing because threads whose policy fails the mask comparison are reset to timesharing; the alternative of a default policy per processor set is more complicated, and the alternative of setting the thread to some allowed policy is potentially unpredictable. The interface routines that manipulate the mask operate by enabling or disabling one policy at a time. The Mach kernel routines that deal with policy and priority are documented in appendix C.

The fixed priority policy is implemented by suppressing the usage adjustments of the timesharing scheduler. Absolute preemption occurs between priority levels, and round-robin scheduling occurs within each priority level. Preemption may be delayed by up to a clock interrupt period on a multiprocessor because interprocessor interrupts are not currently used for preemption. The fixed priority policy allows each thread to be given a quantum for use in the round-robin scheduling within a priority level. This quantum is given to the thread every time it begins to run, including resumption after preemption by a higher priority thread.

Related Work

Previous work on *policy mechanism separation* has proposed separating the scheduler into two pieces: mechanisms implemented in the operating system, and policy decisions made by *policy module*, usually placed in user mode as part of the application [6,10]. This work only considered the problem of scheduling within individual applications and

encountered serious problems in the areas of policy module complexity and decision-making overhead. Mach's processor allocation facility uses policy mechanism separation to address a different problem, scheduling among competing applications. This avoids the problems encountered in prior uses because processor allocation decisions are made infrequently enough to effectively amortize the overhead of boundary crossing costs, and because the complex policy implementation resides in a server that is implemented once for a system rather than a module that must be customized to each application.

Another body of related work concerns the area of *coscheduling*, multiprocessor scheduling policies that attempt to schedule components of an application at the same time, but make no guarantees of success. These policies were originally proposed for medium grain parallel message passing applications (hundreds to thousands of instructions between interactions) that benefit from coscheduling but can achieve reasonable performance in its absence. The major work on coscheduling was done for the Medusa operating system on Cm* [4]. The implementation used a matrix with columns indexed by processors and rows indexed by time; each single-threaded Medusa task occupied one cell in the matrix. Periodic clock interrupts caused each processor to proceed to the next cell in its column; if that cell was empty, it would search other cells in its column to find a task. To achieve effective coscheduling, the processors must advance to the next row in the matrix almost simultaneously. Cm* supported this via synchronized clock interrupts; the periodic clock interrupts for each processor were generated by an interrupt source (a line clock) that was phase-locked to the 60Hz power supplied by the local electric company. Each processor would therefore take clock interrupts and proceed to the next row of the matrix at almost the same time. A second characteristic of Cm* that this work depended on was Cm*'s limited-memory NUMA architecture that essentially precluded load balancing. Because processors took interrupts almost simultaneously, they could never look at other columns in the matrix when searching for a task to run, and given Cm*'s architecture there was no reason to ever do so. In contrast, UMA shared memory machines benefit from short term load balancing, as do the multiprocessor clusters on many current NUMA machines. Synchronized clocks make short term load balancing more difficult and expensive to implement because they preclude the use of a shared data structure for contention reasons.

The Alliant Concentrix scheduler described by Jacobs [5] is an example of an alternative approach to processor allocation. This scheduler supports a fixed number of scheduling classes and uses a *scheduling vector* for each processor to indicate which classes should be searched for work in what

order. Each processor cycles through a set of scheduling vectors based on time durations associated with each vector, typically fractions of a second. Processes are assigned to scheduling classes by their characteristics or by a system call available to privileged users and applications. This scheduler is designed to divide processors among statically defined classes of applications over short periods of time, and contrasts with the Mach orientation of dedicating processors to applications over longer periods of time. Mach's processor sets can be created dynamically as opposed to the fixed number of scheduling classes. Scheduling servers could be implemented by reserving some scheduling classes for their exclusive use, but the static class and vector definitions appear to restrict the flexibility available in forming sets of processors. The Concentrix scheduler also enforces a more restrictive version of gang scheduling in which a blocking operation by any thread blocks the entire gang. This restricts it to applications that do not use system concurrency and makes parallel handling of blocking operations such as I/O and page faults all but impossible.

The notions of multiple scheduling policies and fixed priority scheduling are neither unique to this work or recent discoveries. The StarOS system for Cm* supported multiple scheduling policies, but selected them on a per processor rather than a per thread basis when the system was booted [6]. These features of StarOS saw little use, as the main thrust of the research on CM* was parallel processing in which tasks and processors were explicitly bound on a one-to-one basis, obviating the need for scheduling policies. Fixed priority scheduling has been added to many versions of Unix for real time support, often by expanding the priority range and designating some of the priorities as fixed instead of timesharing [7]. This approach makes the policy implicit in the priority, and is much less flexible and extensible than the explicit policy approach used by Mach.

Conclusion

This paper has described Mach scheduling features that support the explicit resource allocation required by parallel, real time, and other environments. Mach's processor allocation facility supports dedicating processors to applications, and provides a flexible architecture to accommodate different parallel programming models and requirements. The support for non-timesharing scheduling policies allows Mach to incorporate scheduling policies for real time and related environments, with provisions for adding additional policies as needed. The facilities described in this paper were added to Mach after the 2.5 release. They are available in Mach 3.0 (microkernel system), OSF/1, Encore's Mach for the Multimax, and other systems. Further details on the Mach scheduler, `cpu_server`, and the facilities described in this paper can be found in [1, 2].

References

- [1] David L. Black, 'Scheduling Support for Concurrency and Parallelism in the Mach Operating System', *COMPUTER*, 23(5):35-43, May 1990.
- [2] David L. Black, *Scheduling and Resource Management Techniques for Multiprocessors*, PhD Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. July 1990. Available as Technical Report CMU-CS-90-152.
- [3] Dror G. Feitelson and Larry Rudolph, 'Distributed Hierarchical Control for Parallel Processing', *COMPUTER*, 23(5):65-77, May 1990.
- [4] Edward F. Gehring, Daniel P. Siewiorek, and Zary Segall, *Parallel Processing: The Cm* Experience*, Digital Press, Maynard, MA, 1987.
- [5] Herb Jacobs, 'A User-tunable Multiple Processor Scheduler', *Proceedings, Winter 1986 USENIX Conference*, 183-191, January 1986.
- [6] Anita K. Jones, Robert J. Chansler, Jr., Ivor Durham, Karsten Schwans, and Steven Vegdahl, 'StarOS, A Multiprocessor Operating System for the Support of Task Forces', *Proceedings of the 7th Symposium on Operating Systems Principles*, 117-127, December 1979.
- [7] C. Douglass Locke, 'Scheduling in Real Time', *Unix Review*, 8(9):48-54, September 1990.
- [8] Richard F. Rashid, 'Threads of a New System', *Unix Review*, 4(8):37-49, August 1986.
- [9] Andrew Tucker and Anoop Gupta, 'Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors', *Proceedings of the 12th Symposium on Operating Systems Principles*, 159-166, December 1989.
- [10] William A. Wulf, Roy Levin and Samuel P. Harbison. *Hydra/C.mmp: An Experimental Computer System*, New York: McGraw-Hill, 1981.

David L. Black is a Research Fellow at the Cambridge, MA office of the Open Software Foundation's Research Institute, where he is participating in a joint research project with Carnegie Mellon University and other collaborators on microkernel-based operating system environments. His interests in this project include



microkernel technology, real-time, parallel, and distributed computing. Dr. Black was one of the key developers of the Mach operating system at CMU, from which he received the PhD degree in Computer Science in 1990. He also holds an MS in Computer Science from CMU, and three degrees from the University of Pennsylvania, a BA and MA in Mathematics, and a BSE in Computer Science and Engineering. Dr. Black is a member of ACM, IEEE Computer Society, and Sigma Xi. Author's current address is: Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142 (dlb@osf.org).

Appendix A: A Processor Allocation Kernel Interface

This appendix describes the kernel interface for processor allocation. These primitives are used by applications and allocation servers to perform processor allocation. The primitives are divided into groups based on the objects they manipulate. Only the routines to get the host ports are traps; all other primitives are implemented by a remote procedure call message exchange with the kernel.

This interface exists on all machines, including uniprocessors; a kernel configuration switch allows deletion of the support code for processor allocation. On a uniprocessor, the only processor set that exists is the default processor set; its control port is privileged and not available to non-privileged users. The calls to retrieve information about the processor and the default processor set are useful on all machines.

Host Operations

The following operations manipulate and obtain information about Mach hosts, a Mach host is a machine running a single Mach kernel (uniprocessor or multiprocessor):

host self - Obtain host port.

host priv self - Obtain privileged host port. Caller

must be privileged (e.g., Unix super-user). host processors - Obtain list of processors on host.

host info - Obtain information about the host. Extensible to include machine dependent information.

host kernel version - Obtain the version string for the kernel running on host. This is more descriptive than a version number.

For pure Mach kernels (e.g., Mach 3.0), the host priv self trap does not exist. Instead the privileged host port is inserted into the port space of the first task on the system, which is responsible for all further use and control of the port. Operations to obtain a list of processor sets and control privileges for individual processor sets are listed in the processor set section.

Processor Operations

The following operations manipulate and obtain information about processors:

processor start - Start a processor.

processor exit - Exit a processor.

processor control - Send a machine-dependent command to a processor.

processor info - Obtain information about a

processor. Extensible to allow the definition of machine-dependent information flavors.

Support for the start, exit, and control calls is machine-dependent; they have no effect if not supported. The control call is useful for performing console functions on multiprocessors that have one logical console per processor rather than a single system console (e.g., VAXes).

Processor Set Operations

The following operations create, destroy, manipulate, and obtain information about processor sets:

processor set create - Create a new processor set.

processor set destroy - Destroy a processor set. Any assigned processors, tasks, or threads are reassigned to the default processor set. The default processor set cannot be destroyed.

processor set default - Identify the default processor set.

processor set info - Obtain information about a processor set.

processor set tasks - Obtain a list of the tasks assigned to a processor set.

processor set threads - Obtain a list of the threads assigned to a processor set. host processor sets - Obtain a list of all processor sets on a host.

host processor set priv - Obtain control privileges for an individual processor set. This requires the privileged host port.

There is no significance to the ordering of items in any list returned by these operations (processor set tasks, processor set threads, host processor sets). Identification of an object in any of these lists requires an additional comparison, or the use of the appropriate information retrieval (info) operation.

Execution Control Operations

The following operations control the assignment of processors, threads, and tasks to processor sets:

processor assign - Assign a processor to a processor set. Both synchronous and asynchronous assignments are available.

processor get assignment - Find out which processor set a processor is assigned to.

thread assign - Assign a thread to a processor set.

thread assign default - Assign a thread to the default processor set.

thread get assignment - Find out which

processor set a thread is currently assigned to.

task assign - Assign a task to a processor set. Optionally assign all threads within the task.

task assign default - Version of task assign that assigns to the default processor set.

task get assignment - Find out which processor set a task is assigned to.

The default versions of task and thread assignment are necessary because unprivileged applications are not allowed to control scheduling in the default processor set, and therefore cannot execute assignment operations with it as a target. The assignment of a task is used only to initialize the assignment of new threads created in it.

Appendix B: Cpu server Interfaces

This appendix documents the interfaces to the Mach `cpu_server`. Both the low level remote procedure call interface and the library interfaces built on top of it are included. It is important to emphasize that this is an interface to a server implementing a specific allocation policy for UMA multiprocessors. Servers that implement other policies and/or support different architectures will have different interfaces.

Server RPC Interface

This section describes the basic remote procedure call (RPC) interface used to access the server. This interface is based on the concept of a request object that is created and manipulated by the following primitives:

`cpu_server info` - Get information about the server.

`cpu request create` - Create a new request for processors.

`cpu request add` - Add a processor set, number of processors element to a request.

`cpu request set notify` - Request notifications and provide a port to which they will be sent.

`cpu request activate` - Indicate that the request is assembled, and ask the server to find processors for it.

`cpu request destroy` - Destroy a request.

`cpu request status` - Get status of a request for processors.

The first two calls are made on a generic service port exported by the server; it is registered with the local nameservice under the name `'cpu_server'`. There are three options available for the activate call:

Destroy - Destroy the processor sets when the

request is completed. This is intended to support naive users by preventing a program that overruns its request from going into suspended animation; destroying its processor sets forces the program back into the default processor set where it will continue to run.

Repeat - Repeat the request for a longer period of time. This supports long requests without excluding shorter requests.

Task - Informs the server that the application using the processor sets is a task, and identifies the task. This allows the server to optimize assignment of processors by suspending the task because processor assignment is faster if the processors are idle. This option is requested by using `cpu request activate task` instead of `cpu request activate`.

Library Interfaces

The server interface and the kernel interface for processor allocation will be used directly by programs that require explicit control over which threads are executing on which processors at which time. Most applications have less stringent processor allocation requirements, and can therefore use simpler library interfaces that hide all of the internal details of interaction with the kernel and server. Four such interfaces have been developed: *allocate*, *task*, *hook*, and *task-hook*. All of the interfaces are independent; processors must be deallocated with the `deallocate` call from the interface that was used to allocate them.

The *allocate* interface supports a single allocation of a pool of processors. It exports the `allocate processors` and `deallocate processors` calls. `allocate processors` does not return until the allocation of processors has started; it also performs a task assignment so that the initial thread and all threads and tasks subsequently created share the allocated processors. If a program overruns its time allocation, it will continue to run, but without dedicated processors. `deallocate processors` frees the allocated processors. It must be called by a thread in the same task that did the allocation.

The *task* interface is identical to the *allocate* interface, but is restricted to applications consisting of a single task so that the server can exploit efficiencies available in this case (suspending the task before removing processors). The task interface exports the `task allocate processors` and `task deallocate processors` calls.

The *hook* interface supports allocation of a pool of processors, with user scheduling hooks. It exports the `allocate processors with hooks` and `deallocate processors with hooks` calls. The allocation call defines two scheduling hooks, a *start hook* and an *end hook*. The *start hook* is called after the processors are allocated, and the *end hook* is called

approximately 1 second before processor deallocation. A thread must be dedicated to the allocate processors with hooks call; this means that the calling thread does not return until the allocation has ended. The dedicated thread is assigned to the allocated processors (this can be reversed by explicitly assigning it as part of the *start hook*). All threads within its task and subsequently created tasks are also assigned to the allocated processors. Both *start hook* and end hook must return for the interface to function correctly. In particular, the interface will break if *end hook* does not return before the processors are deallocated.

Finally, there is the *task-hook* interface, which combines the functionality of the hook interface with the server optimization of the task interface; the calls in this interface are the hook interface calls with *task* prefixed.

Appendix C: Priority and Policy Kernel Interface

This appendix describes the kernel interface for scheduling policies and priorities. Only timesharing and fixed priority policies are currently implemented, but the policy interface is extensible to allow the definition and use of other policies in the future. The timesharing policy is for the usual timeshared use of a machine. The fixed priority policy is intended to support soft real time applications.

Priority Operations

The following operations manage scheduling priorities:

thread priority - Set a thread's priority and (optionally) its maximum priority. This call can only lower the maximum priority.

thread max priority - Set a thread's maximum priority to any value. Requires scheduling control privilege for the processor set to which the thread is assigned.

task priority - Set a task's priority and (optionally) the priorities of all threads in it.

processor set max priority - Set the maximum priority for a processor set and (optionally) all threads that are assigned to it.

Policy Operations

The following operations control the use of scheduling policies:

thread policy - Set scheduling policy for a thread.

processor set policy enable - Enable use of a scheduling policy for a processor set.

processor set policy disable - Disable use of a scheduling policy for a processor set. Optionally reset any threads using it to the timesharing policy.

A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility

Marc Guillemont, Jim Lipkis, Doug Orr, Marc Rozier – Chorus systemes

ABSTRACT

An important trend in operating system development is the restructuring of the traditional monolithic OS kernel into independent servers running on top of a minimal nucleus or microkernel. This approach arises out of the need for modularity and flexibility in managing ever-growing complexity caused by new functions and new architectures. In particular, it provides a solid architectural basis for distribution, fault tolerance, and security. Microkernel-based operating systems have been a focus of research for a number of years, and are now beginning to play a role in commercial UNIX systems.

However, the ultimate feasibility of this attractive approach is not yet widely accepted. A primary concern is efficiency: can a microkernel-based modular OS provide performance comparable to that of a monolithic kernel – at least when running on a monolithic architecture? The elegance and flexibility of the client-server model may exact a cost in message-handling and context-switching overhead. If this penalty is too great, commercial acceptance will be limited. Another pragmatic concern is compatibility: in an industry relying increasingly on portability and standardisation, compatible interfaces are needed not only at the level of application programs, but also for device drivers, streams modules, and other components. In many cases binary as well as source compatibility is required. These concerns affect the structure and organisation of the operating system.

The Chorus team has spent the past six years studying and experimenting with UNIX kernelisation as an aspect of its work in modular distributed and real-time systems. In this paper we examine aspects of the current CHORUS system in terms of its evolution from the previous version. Our focus is on pragmatic issues such as performance and compatibility, as well as considerations of modularity and software engineering.

Microkernel architectures

A recent trend in operating system development consists of structuring the OS as a modular set of system servers sitting on top of a minimal microkernel, rather than using the traditional monolithic structure. This new approach promises to help meet systems and platform builders' needs for a sophisticated OS-development environment that can cope with growing complexity, new architectures and changing market conditions. In this OS architecture, the microkernel provides system servers with generic services independent of a particular operating system, such as processor scheduling and memory management. The microkernel also provides a simple Inter-Process Communication (IPC) facility that allows system servers to call each other and exchange data independently of where they are executed in a multiprocessor, multicomputer or network configuration.

This combination of primitive services forms a standard base which in turn supports the implementation of functions that are specific to a particular operating system or environment. These system-specific functions can then be configured as

appropriate into system servers managing the other physical and logical resources of a computer system, such as files, devices and high-level communication services. Such a set of system servers is called a *subsystem*. Real-time systems tend to be built along similar lines, with a very simple generic executive supporting application-specific real-time tasks.

UNIX and Microkernels

UNIX introduced the concept of a standard, hardware-independent operating system, whose portability allowed platform builders to reduce their time to market by obviating the need to develop proprietary operating systems for each new platform.

However, several trends are pulling UNIX away from its roots. As more function and flexibility is continually demanded, it is unavoidable that today's versions should be increasingly more complex. For example, UNIX is being extended with facilities for real-time applications and on-line transaction processing. Even more fundamental is the move toward distributed systems. Today's computing environments require that new hardware and software resources, such as specialised servers and

applications, be integrated into a single system, distributed over some kind of communication medium. The range of communication media commonly encountered includes shared memory, buses, high speed networks, local and wide-area networks. This trend will become fundamental as collective computing environments emerge to better map natural human organisation.

For these reasons, it is desirable to map UNIX onto the microkernel architecture, where machine-dependencies may be isolated from unrelated abstractions and facilities for distribution are incorporated at a very low level.

The attempt to reorganise UNIX into the framework of a microkernel architecture poses problems, however, since the resultant system must produce the same set of behaviours found in traditional UNIX. A primary concern is efficiency: a microkernel-based modular OS must provide performance comparable to that of a monolithic kernel. The elegance and flexibility of the client-server model may exact a cost in message-handling and context-switching overhead. If this penalty is too great, commercial acceptance will be limited. Another pragmatic concern is compatibility: in an industry relying increasingly on portability and standardisation, compatible interfaces are needed not only at the level of application programs, but also for device drivers, streams modules, and other components. In many cases binary as well as source compatibility is required. These concerns affect the structure and organisation of the operating system.

There is work in progress to port UNIX to a microkernel architecture on a number of fronts, including the Mach [Golub 90, Cheriton 90], and Amoeba [Tanenbaum 89] projects. CHORUS versions V2 and V3 represent the work we have done to solve these problems.

The CHORUS microkernel technology

The Chorus team has spent the past six years studying and experimenting with UNIX kernelisation as an aspect of its work in modular distributed and real-time systems. The first implementation of a UNIX-compatible microkernel-based system was developed between 1984 and 1986 as a research project at INRIA. Among the goals of this project were to explore the feasibility of shifting as much function as possible out of the kernel, and to demonstrate that UNIX could be implemented as a set of modules that did not share memory. In late 1986 a new version, based on an entirely rewritten CHORUS nucleus, was launched at Chorus systèmes. The current version shares most of the goals of the previous and adds some new ones, including real-time support and – not incidentally – commercial viability. A UNIX subsystem compatible with System V Release 3.2 is currently available, with System V Release 4.0 and

BSD under development.

In this paper we examine aspects of the current CHORUS system in terms of its evolution from the previous version. Our focus is on pragmatic issues such as performance and compatibility, as well as considerations of modularity and software engineering. The earlier version was built around a pure message-passing model, in which strict protection was incorporated at the lowest level. In contrast, the goal in the current system is to provide nucleus primitives which are as lightweight and flexible as possible. It is left to the subsystem designer to negotiate the tradeoffs between simplicity and efficiency, on the one hand, and more sophisticated function or greater protection, on the other. For example, separate subsystem servers may be either isolated in separate address spaces or (on a single site) share address spaces, and may communicate with user applications strictly through messages or by handling hardware traps. Further examples of evolution are found in the area of inter-process communication. The earlier system adopted a communication-based execution model resembling atomic transactions; this was replaced by the remote procedure call (RPC) paradigm which has since evolved into a very efficient lightweight RPC protocol. Issues of software engineering, UNIX support, and performance were involved in various stages of this progression.

The System V Release 3.2 implementation performs comparably with well-established monolithic kernel systems on the same hardware, and better in some respects. As a testament to commercial viability, the system has been adopted for actual use in commercial products ranging from X terminals and telecommunication systems to mainframe UNIX machines.

In section 2, we overview the previous CHORUS version. Section 3 summarises the main design decisions for the current version. The next 2 sections focus on specific aspects of the current design.

CHORUS V2 Overview

The CHORUS project, while at INRIA, began researching distributed operating systems with CHORUS V0 and V1. These proved the viability of a modular message-based distributed operating system, examined its potential performance, and explored its impact on distributed applications programming.

Based on this experience, CHORUS V2 [Armand 86, Rozier 87] was developed. It represented the first intrusion of UNIX into the peaceful CHORUS landscape. The goals of this third implementation of CHORUS were

1. To add UNIX emulation to the distributed system technology of CHORUS V1;
2. To explore the outer limits of kernelisation; demonstrate feasibility of a UNIX implementation

with a minimal kernel and semi-autonomous servers;

3. To explore the distribution of UNIX services;
4. And to integrate support for a distributed environment into the UNIX interface.

Since its birth, the CHORUS architecture has always consisted of a modular set of servers running on top of a microkernel (nucleus) which included all of the support necessary for distribution.

The basic execution entities supported by the V2 nucleus were mono-threaded *actors* running in user mode, isolated in protected address spaces. Execution of actors consisted of a sequence of processing-steps which mimic atomic transactions: ports represented operations to be performed; messages would trigger their invocation and provide arguments; the execution of remote operations were synchronised at explicit commit points. A permanent concern in the design of CHORUS is that fault-tolerance and distribution are tightly coupled; hardware multiplication increases the probability of faults, while hardware redundancy gives a better chance to recover from these faults.

Communication is, as in many systems, based on exchange of messages through *ports*. Ports are attached to actors, and have the capability to migrate from one actor to another. Ports can be gathered into *port groups*, allowing message broadcasting as well as functional addressing. The port group mechanism provides a flexible set of client-server mapping semantics, including for example dynamic reconfiguration of servers.

For performance reasons, message contents have been uninterpreted by the kernel (untyped) in both versions V2 and V3.

Lesson: A guideline in the design of CHORUS V2, retained in V3, is to avoid forcing simple and efficient applications to pay the burden of sophisticated mechanisms required only by some specific classes of programs.

Ports, groups and actors were given global unique names, built in a distributed fashion by each Nucleus for use by system entities. Private, context-dependent names were exported to user programs. These port descriptors were inherited in the same fashion as file descriptors by UNIX processes.

Lesson: Most of the CHORUS design intends to give sites as much autonomy as possible, in particular through distributed algorithms. Autonomy favours simplicity and robustness.

The context-dependent names were provided for security and ease of use. It was difficult, however, for applications to exchange port names, since it required intervention of the nucleus and posed bootstrapping problems. As a result, context-dependent names were

inconvenient for distributed applications, such as name servers. In addition, many applications had no need of the added security the context-dependent names provided. The standard way to name objects in V3 is through global names.

UNIX

On top of this architecture, a full UNIX System V was built.

In V2, the whole of UNIX was split into three servers: a Process Manager, dedicated to process management, a File Manager for block device management and a Device Manager for character device management. UNIX network facilities (sockets) were not implemented at this time. In addition, the nucleus was complemented with two servers, managing ports and port groups, and remote communications, respectively.

Lesson: A goal of the V2 project was to determine what were the minimal set of functions that a microkernel should have in order to support a robust base of computing. To that end, the management of ports and port groups was put into a server external to the nucleus.

Providing the ability to replace a fundamental portion of the IPC did not prove to be useful, since IPC was a fundamental and critical element of all nucleus operations. Maintaining it in a separate server rendered it more expensive to use. Port and port group management was moved back into the nucleus for V3.

A UNIX process was implemented as an actor. All interactions of the process with its environment, i.e. all system calls, were performed as exchanges of messages between the process and system servers. Signals were implemented as messages, also.

This modularisation of UNIX impacted it in the following way:

1. UNIX data structures were split (and not duplicated, in order to avoid consistency problems) between the nucleus and several servers. Messages between these servers contained the information managed by one server and required by another in order to provide its service (through the use of clever splitting techniques, the amount of information required can be minimised).
2. Most UNIX objects (files, in particular) were designated by network-wide capabilities which could be exchanged freely between servers and sites – this proved to be key to providing distributed UNIX services.

The context of a process contained a set of capabilities for the objects accessed by the process.

As much of the UNIX system calls as possible were implemented by a process-level library. The process context (mainly a set of file capabilities) was stored in process-specific library data. The library invoked the servers when necessary (i.e. the Process Manager for `fork(2)`, the file manager for `read(2)`, etc.), using the RPC facility.

This library offered source-level compatibility with UNIX, only (binary compatibility was not included in the goals of the project). The library resided at a predefined user virtual address in a write-protected area. Library data holding the process context information was not completely secure from malicious or unintentional modification by the user. Thus, errant programs could experience new, unexpected error behaviour. In addition, programs that depend on the standard UNIX address space layout could cease to function because of the additional address space contents.

Lesson: For 100% UNIX compatibility, it is necessary to maintain the standard UNIX trap interface and address space layout. Use of shared libraries can produce compatibility and error-detection problems.

Lesson: Implementing functionality in user-level servers imposed message passing and context switch overheads not present in the same implementation found in a traditional, monolithic kernel. In V3, these new overheads were offset or compensate for through other means such as the use of supervisor actors (see Section 4).

The V2 process model provided most naturally for single-threaded, synchronous model of process execution. To treat asynchronous signals, it was necessary to introduce the concept of priorities within messages to expedite the invocation of a signaling operation. Even so, the priorities went into effect only at fixed synchronisation points, making it impossible to exactly represent UNIX signal behaviour. Further work has shown that signals are one of the stumbling blocks for building fault tolerant UNIX systems.

Lesson: While elegant, the processing-step model of computation was a poor fit with the asynchronous signal model of exception handling. In order to provide high-quality UNIX emulation, a more general computational model was necessary for CHORUS V3.

Extension of UNIX

CHORUS V2 extended UNIX in two ways:

- UNIX services were extended to allow distribution (e.g. remote process creation, remote file access)

while retaining their original interface.

- access to new services (e.g. IPC) was provided without breaking the UNIX semantics.

Distribution of UNIX services

The modularity of CHORUS's UNIX and its inherent protocols permits a first simple extension to UNIX: for example, a process may access a remote file server exactly in the same way as a local one, as it relies on IPC which can cross machine boundaries; the location-transparent interface of the IPC brings its transparency at the user-level file interface.

In addition, CHORUS V2 extended the UNIX file semantics with *port nodes* which allow any server able to process file system calls to be designated with a symbolic pathname. This is used to automatically interconnect file trees. Distributed file protection was also explored.

For processes, new protocols between Process Managers were developed in order to distribute `fork` and `exec` operations. This is facilitated by the fact that

- the entire process context is managed by one single system server (Process Manager);
- this context contains only global references to resources (capabilities).

Therefore, creating a remote process can be done almost entirely by transferring the process context from one Process Manager to another.

Since signals were implemented as messages, their distribution comes for free given that processes have global PIDs.

Introduction of new services

The major new service introduced at user-level was the CHORUS IPC. Its UNIX interface was designed in the standard UNIX style:

1. Ports and port groups were known, from within processes, by local identifiers. Access to a port was controlled analogously to the access to a file.
2. Ports and port groups were protected similarly to files (with *uids* and *gids*).
3. Port and port group access rights were inherited on `fork` and `exec` exactly as are file descriptors.

Lessons. We went too far in this direction. Introduction of IPC in the user-level interface was important, but it did not need to maintain the UNIX style.

Employing the same form as the UNIX file descriptor for port descriptors was intended to provide uniformity of model. The semantics of ports were sufficiently different from the semantics of files to negate this advantage. In operations such as `fork`, for

example, it did not make sense to share port descriptors in the same fashion as file descriptors. Attempting to force ports into the UNIX model resulted in confusion.

CHORUS V3

New goals

The design of CHORUS V3 system [Armand 89, Armand 90, Herrmann 88, Rozier 88] has been strongly influenced by a new major goal: to design an operating system technology suitable for the implementation of commercial operating systems. CHORUS V2 was a UNIX-compatible distributed operating system. CHORUS V3 is a distributed microkernel able to support different operating systems, as sets of subsystems, compatible with operating system standards while meeting the new needs of commercial systems builders.

These new goals determined new guidelines for the design of the CHORUS V3 technology:

- **Portability:** the CHORUS V3 microkernel must be highly portable on various machine architectures. In particular, this motivated the design of an architecture-independent memory management system [Abrossimov 89], taking the place of the hardware-specific CHORUS V2 memory management.
- **Generality:** the CHORUS V3 microkernel must provide a set of functions which are sufficiently generic to allow the implementation of various operating system process semantics. This motivated a move from the restrictive (though powerful) event-driven CHORUS V2 actor model to a more general multi-thread model. Similarly, some UNIX-related features had to be removed from the CHORUS V2 kernel.
- **Compatibility:** UNIX source compatibility in CHORUS V2 had to be extended to binary compatibility in V3, both for user applications and device drivers. In particular, the CHORUS V3 kernel had to provide tools allowing subsystems to build binary compatible interfaces. In addition, the CHORUS V3 processing model required an easy (and efficient) implementation of complex process semantics, including asynchronous signal delivery.
- **Real-time:** process control and telecommunication systems comprise important targets for distributed systems. In this area, the responsiveness of the system is of prime importance. The CHORUS V3 kernel is, first and foremost, a distributed real-time executive. The real-time features may be used by any subsystem, allowing for example a UNIX subsystem to be naturally extended to be suitable for real-time applications needs.
- **Performance:** for commercial viability, good

performance is essential in an operating system. While offering the base for building modular and well-structured operating systems, the kernel interface must allow these operating systems to reach at least the same performance as conventional, monolithic, implementations.

Architectural elements retained from CHORUS-V2

Outside of the transaction-style processing model, most of the architectural elements of CHORUS V2 were retained in CHORUS V3.

CHORUS V2 basic IPC abstractions (location transparency, untyped messages, asynchronous and RPC protocols, ports and port groups) have proven to be very well suited to the implementation of distributed operating systems and applications. These abstractions have been entirely retained for CHORUS V3. However, the interface and implementation of the IPC facilities have been redesigned.

In addition, the basic UNIX subsystem modular architecture has been retained in the implementation of CHORUS V3 UNIX subsystems. Some new servers (such as the Socket Manager) have been added for new function not included in CHORUS V2.

New CHORUS V3 elements

Although the main CHORUS principles were retained in the new version, some important enhancements have been made:

- The event-driven mono-thread CHORUS V2 processing model was discarded, for a more general multi-thread model. A CHORUS V3 actor is merely a resource container, offering in particular an address space in which multiple threads may execute. Threads are scheduled as independent entities, allowing for example real parallelism on a multiprocessor architecture. In addition, multiple threads allowed the simplification of the control structure of server-based applications. New kernel services, such as thread execution control and synchronisation have been introduced.
- CHORUS V3 makes the port and group global names (Unique Identifiers) visible to the user, discarding the UNIX-like CHORUS V2 contextual naming scheme. The first consequence is simplicity: port and groups names may be freely exchanged by kernel users, avoiding the need for the kernel to maintain complex actor context. The second consequence is a lower level of protection: the CHORUS V3 philosophy is to provide subsystems with the means for implementing their own level and style of protection rather than enforcing protection directly in the microkernel.

The kernel must be able to maintain its simplicity and efficiency for users or subsystems (e.g. real-time applications) which do not require high level services.

- In CHORUS V2, the low-level device drivers were implemented within the kernel. Physical resource managers made use of these low-level functions by the means of IPC requests. For example, a file manager received disk controller interrupts by the means of IPC messages. Although this model was very clean, it had serious drawbacks:
 - The kernel needed to be modified each time a new device type was to be supported on the machine.
 - The interrupt response time was quite long, forcing parts of highly reactive real-time applications to be partially implemented in the kernel itself.

In CHORUS V3, all device drivers are implemented outside the kernel, as supervisor actors (see Section 4) which are able to directly handle hardware interrupts with minimal latency (only a few microseconds), and to contain threads which may access protected hardware resources. In addition to keeping the kernel simple and efficient, this model is very well adapted to the dynamic loading of device drivers. The supervisor actor is one of the main new building blocks of CHORUS V3 (for a full description, see Section 4).

- Finally, some structural modifications have been made: the CHORUS V3 kernel fully handles ports, groups and actors, which were managed, in CHORUS V2, by a cooperation of the kernel, the port/group manager and the process manager. This change was driven by the observation that ports, actors and groups are basic kernel abstractions. Splitting their management did not provide significant benefit, but did impact system performance.

As a consequence of this kernel evolution, the UNIX subsystem implementation has evolved. In particular, full UNIX binary compatibility was achieved. Internally, the UNIX subsystem makes use of new kernel services, such as multi-threading and supervisor actors. The CHORUS V2 user-level UNIX system-call library has been moved inside the Process Manager, now invoked by traps.

In the next sections, we focus on some of these new elements which impact our two main goals: compatibility and performance.

Evolution in nucleus support for subsystems: Supervisor Actors

Supervisor actors are actors which share the kernel address space and whose threads execute in a privileged machine state (which usually implies the ability to execute privileged instructions, etc.). Otherwise, supervisor actors are fundamentally very similar to regular user actors. They may create multiple ports and threads, and their threads access the

same nucleus interface. Any user program can be run as a supervisor actor, and any supervisor actor which does not make use of privileged instructions or *connected handlers* (see below) can be run as a user actor, in both cases without recompiling the program. (A relink is necessary.) Although they share the kernel address space, supervisor actors are paged just as user actors and may be dynamically loaded and deleted.

Supervisor actors alone are granted direct access to the hardware event facilities. Using a standard kernel interface, any supervisor actor may dynamically establish a handler for any particular hardware interrupt, system call trap, or program exception. A connected handler executes as an ordinary subroutine, called directly from the corresponding low-level handler in the kernel. Several arguments are passed, including the interrupt/trap/exception number and the processor context of the executing thread. The handler routine may take various actions, such as processing an event and/or awakening a regular thread in the actor, and then returns to the kernel.

It is important to note that no subsystem in CHORUS V3 is ever *required* to use connected handlers or supervisor actors. A subsystem designer may choose to export a programming interface based on messages rather than traps, for example. The CHORUS kernel can handle program exceptions with an RPC message sent to a designated exception port, if desired, rather than calling a connected exception handler. If the subsystem includes device drivers, then it is necessary to process device interrupts. Even this can be done in user mode actors if desired, using a stub supervisor actor to translate interrupts into messages. However, connected handlers provide significant advantages in both performance and binary compatibility.

External device drivers

Connected interrupt handlers allow device drivers to exist entirely outside of the kernel, and to be dynamically loaded and deleted, with no loss in interrupt response or overall performance. Interrupt handlers may be stacked, since multiple device types often share a single interrupt level. In this case the sequence of handlers is executed in priority order until one of them returns a code indicating that no further handlers should be called. Connected interrupt handlers have been designed to allow subsystems to incorporate proprietary, object-only device drivers that conform to one of the relevant binary standards that are emerging in this area. Without this mechanism, object compatibility would require incorporating entire device drivers in the kernel.

Compatibility

System call trap handlers are also essential for both performance and binary compatibility. Any subsystem may dynamically connect either a general trap-handling routine or a table of specific system call handlers, the latter providing an optimised path for UNIX-style interfaces. An alternative mechanism, the system-wide user-level shared library used in CHORUS V2, would seem to provide equivalent system call performance. However, as we have seen, it is difficult to protect subsystem data that share the address space of the user program, especially if processes are multi-threaded. As we have seen, malicious or innocent but erroneous programs can change the behaviour of system calls. If functions must be moved from the shared library into separate servers for protection, increased IPC traffic results. Finally, the presence of the library code and data in the user context can interfere with binary programs that use a large portion of the address space or manage the address space in some particular fashion. Traps to supervisor actors, by contrast, provide a low-overhead, self-authenticating transfer to a protected server, while maintaining full transparency for the user program.

Performance benefits

Performance benefits of supervisor actors come in several areas. Memory and processor context switches are minimised through use of connected handlers rather than messages, and in general through address-space sharing of actors of a common subsystem which happen to be running on a single site. Trap expense can be avoided for nucleus system calls executed by supervisor actors. Finally, supervisor actors allow a new level of RPC efficiency. The lightweight RPC mechanism of [Bershad 90] optimises pure RPC for the case where client and server reside on the same site. We further optimise for the case where no protection barrier need be crossed between client and server. This featherweight RPC is substantially lower in overhead, while still mediated by the kernel and still using an interface similar to that of pure RPC.

Construction of subsystems

Subsystems may be constructed using combinations of supervisor or user actors. Any server may itself belong to a subsystem, such as UNIX, as long as it does not produce any infinite recursions, and may be either local or remote. Servers that need to issue privileged instructions or that are responsible for handling traps or interrupts must be supervisor actors.

Protection issues

Computer systems often give rise to tradeoffs between safety and performance, and we must consider the nature of the sacrifice being made when multiple servers and the microkernel share the supervisor address space. Protection barriers are weakened, but only among mutually-trusted programs, i.e., servers within a single subsystem. First, a strong design rule, which must be strictly followed, is that servers must *never* export themselves the address-space sharing: this feature is only used by the kernel in order to optimise servers invocations. Allowing a server to explicitly access other servers data would totally break the system modularity. This being enforced, the only genuine sacrifice is a degree of bug isolation among the components of a running system. This is somewhat mitigated by the fact that subsystem servers may be debugged in user mode. In fact, this forms our day-to-day development activity: servers are developed, debugged in user mode; when validated, they are loaded as supervisor actors for better performance, if necessary. However, the overall CHORUS philosophy is to allow the subsystem designer or even a system manager to choose between protection and performance on a case-by-case basis, and to alter those choices easily.

Evolution in IPC

CHORUS V3 IPC is based on the accumulated experience gained since V0. Here again, the main characteristics of the IPC facilities are their simplicity and performance.

The first aspect which has evolved since V2 is naming: for many reasons, distributed applications need to transfer names among their individual components. This is most efficiently achieved with a single space of global names that are usable in any context, from kernel to application level. The main difficulty with this style of naming is protection.

In CHORUS V3, ports and port groups have global names (Unique Identifiers) which are visible at every level. Basic protection for these names is threefold:

1. All messages are stamped by the nucleus with the unique identifiers of the sending actor and port, and with a *protection identifier* associated with the port. (Protection identifiers may be modified only by trusted actors.) Thus subsystems may implement their own user authentication mechanisms.
2. Global names are randomly generated in a large name space; knowing a valid global name does not help much in finding other valid names.
3. Objects within CHORUS may be named using capabilities which consist of a <name, key> tuple. Capabilities are constructed using whatever techniques are deemed appropriate by the server that provides them, and may incorporate protection schemes.

Port groups, as implemented by the Nucleus, have keys which are related to the group name by means of a non-invertible function. Knowledge of the group name conveys the right to send messages to the group, but knowledge of the key is required to insert or delete members from the group.

Higher degrees of port and/or message security can be implemented by individual subsystems, as required. Subsystems may act as intermediaries in message communications to provide protection, or may choose to completely exclude IPC from the set of abstractions they expect to user tasks.

A second area of evolution in the CHORUS V3 IPC is message structure.

The memory management units of most modern machines allow moving data from the address space of one actor to the address space of another actor by remapping. This facility is exploited in CHORUS V3 IPC, which allows transmission of message bodies between actors (within a single site) by means of address remapping. In situations where data is to be copied and not moved between address spaces, CHORUS V3 has copy-on-write facilities that allow the data to be efficiently transferred only as needed. The typical communication that makes use of this facility involves the exchange of a large amount of data (e.g. I/O operations).

It is often the case that messages contain a (large) data area, accompanied by some auxiliary information such as a header or some parameters (e.g. pathname, size, result of I/O, etc.). Frequently, the auxiliary information is physically disjoint from the primary data. In CHORUS V2, assembling these two discontinuous fragments into a single message required that extra copying be done by the user.

CHORUS V3 splits message data into two parts:

- a message body, which has a variable size and may be copied or moved; typically contains the raw data;
- the message annex, which has a fixed size and is always copied; it typically contains the associated parameters or headers.

This division also allows one software layer to provide data, while another provides header or parameter information. For example, the V3 implementation of the `write` system call receives the address of a data buffer from the caller; it appends a header describing the data area and sends both to the device responsible for performing the operation.

A third issue is the processing/communication intertwine. The CHORUS V2 execution model was event or communication-driven. In CHORUS V3, the processing model has been inverted – actors are multi-threaded and the basic mechanism for inter-process synchronisation is RPC. Thus, the CHORUS V3 model is much closer to the traditional procedural model of computation. Multi-threading

allows the multiplexing of servers, simplifying their control structure while potentially increasing concurrency and parallelism. RPC is well understood and straightforward to program.

In addition, for applications that require basic, low-level communications, asynchronous IPC is provided. This IPC has very simple semantics – it provides unidirectional communication incorporating location transparency, with no error detection or flow control. Higher-level protocol layers provided by the user or subsystem can be built on top of this minimal nucleus function.

Conclusion

With CHORUS-V2, we experimented with a first-generation microkernel based UNIX system. A UNIX emulation was built as an application of a pure message-based microkernel. Our microkernel approach proved its applicability to building UNIX operating systems for distributed architecture in a research environment.

The challenge of CHORUS-V3 design was to make this technology suitable for commercial systems requirements, i.e. performance and full compatibility. Our second-generation microkernel design was driven by these absolute requirements. It led to reconsider the role of the microkernel. Instead of strictly enforcing a single, rigid, system architecture, the microkernel is now limited to featuring of a set of basic, simple and versatile tools. Subsystem designers have more freedom to define their operating system architecture, selecting the most appropriate tools. Such decisions like choosing between high security and optimal performance or system expandability are not to be enforced a priori by the microkernel.

The CHORUS-V3 microkernel has met its requirements: the CHORUS/MiX microkernel based UNIX system is efficient, fully UNIX compatible while built in a truly modular way. It has been adopted by a number of manufacturers for real-time and distributed commercial UNIX systems.

Further work will concentrate on taking benefit of this technology to provide advanced operating system features, like a distributed UNIX with a single system image and Fault Tolerant UNIX.

References

- [Abrossimov 89] V. Abrossimov, M. Rozier, M. Shapiro. *Generic Virtual Memory Management for Operating System Kernels*. Proc. of 12th ACM Symposium on Operating Systems Principles, Litchfield Park, Arizona (USA), December 1989, pp. 27
- [Armand 86] François Armand, Michel Gien, Marc Guillemont, Pierre Léonard. *Towards a Distributed UNIX System – The CHORUS Approach*. Proc. of EUUG Autumn'86 Conference,

Manchester (UK)

- [Armand 89] François Armand, Michel Gien, Frédéric Herrmann, Marc Rozier. *Revolution 89 or Distributing UNIX Brings it Back to its Original Virtues*. Workshop on Experiences with Building Distributed and Multiprocessor Systems, Ft. Lauderdale (USA), October 1989, pp. 20
- [Armand 90] François Armand, Frédéric Herrmann, Jim Lipkis, Marc Rozier. *Multi-threaded Processes in CHORUS/MiX*. Proc. of EUUG Spring'90 Conference, Munich (Germany), April 1990, pp. 1-13
- [Bershad 90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, Henry M. Levy. *Lightweight Remote Procedure Call*. ACM Transactions on Computer Systems, vol 8, 1, February 1990, pp. 37-55
- [Cheriton 90] David R. Cheriton, Gregory R. Whitehead, Edward W. Sznyter. *Binary Emulation of UNIX using the V Kernel*. Proc. of Summer 1990 USENIX Conference, Anaheim, California (USA), June 1990, pp. 73-86
- [Golub 90] Davic Golub, Randall Dean, Alessandro Forin, Richard Rashid. *UNIX as an Application Program*. Proc. of Summer 1990 USENIX Conference, Anaheim, California (USA), June 1990, pp. 87-96
- [Herrmann 88] Frédéric Herrmann, François Armand, Marc Rozier, Vadim Abrossimov, Ivan Boule, Michel Gien, Marc Guillemont, Pierre Léonard, Sylvain Langlois, Will Neuhauser. *CHORUS, a New Technology for Building UNIX Systems*. Proc. of EUUG Autumn'88 Conference, Cascais (Portugal), October 1988, pp. 1-18
- [Rozier 87] Marc Rozier, José Legatheaux-Martins. *The CHORUS Distributed Operating System: Some Design Issues*. Distributed Operating Systems, Theory and Practice, Berlin, BRD, Springer-Verlag, 1987, pp. 261-286
- [Rozier 88] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, Will Neuhauser. *CHORUS Distributed Operating Systems*. Computing Systems Journal, vol 1, 4, The Usenix Association, December 1988, pp. 305-370
- [Tanenbaum 89] Andrew Tanenbaum, Rob van Renesse, Hans van Staveren. *A retrospective and Evaluation of the Amoeba Distributed Operating System*. Technical Report, Vrije University, Amsterdam (The Netherlands), October 1989, pp. 27

Marc Guillemont is a co-founder and Chorus systèmes' director of engineering. He joined INRIA in 1977 to work on the Cyclades project and was also member of the initial CHORUS research project team in 1980 before he became head of the team. He managed the final research phases of CHORUS before developing the commercial version. Marc Guillemont graduated from Ecole Polytechnique in 1971 and earned a PhD in Computer Science from Grenoble University.



Jim Lipkis is Senior Engineer at Chorus systèmes. At New York University, he was responsible for design and development of operating system and programming language software for highly parallel shared-memory multi-processors, including the NYU Ultracomputer.



Douglas Orr is Senior Engineer at Chorus systèmes. He graduated from the University of Michigan and has worked for Apollo Computer and Carnegie Mellon University. His interests include operating systems, computer networks, and existentialism.



Marc Rozier is the head of the CHORUS distributed microkernel development team within Chorus systèmes. He graduated from "Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble" (ENSIMAG) before earning a PhD in Computer Science from Institut National Polytechnique de Grenoble (INPG). He joined INRIA in 1982 as a researcher in the CHORUS distributed operating system project. He worked on both the design and implementation of two versions of CHORUS. In 1987, he became one of the founders of Chorus systèmes.



Authors may be contacted at:

Chorus systèmes
6, avenue Gustave Eiffel
F-78182 Saint Quentin-en-Yvelines Cedex
France

Tel: +33 1 30 64 82 00

Marc Guillemont <mgu@chorus.fr>

Jim Lipkis <lipkis@chorus.fr>

Doug Orr <doug@chorus.fr>

Marc Rozier <mr@chorus.fr>

Partitioned Multiprocessors and The Coexistence of Heterogeneous Operating Systems

Nick Vasilatos – Concurrent Computer Corporation

ABSTRACT

This presents a variety of issues arising from the α Research Group at Concurrent's effort to develop a means of partitioning an SMP computer system so as to share it between multiple (widely different) operating systems. A simple (minimalistic) mechanism for hosting multiple, (largely unmodified) heterogeneous operating systems and providing a unified programming environment inclusive of the facilities of both, is presented.

Basic motivations for the undertaking are discussed in the context of our research and development effort – primarily aimed at developing α , a new object-oriented, distributed, real-time operating system, which has generated significant secondary requirements for communicating with the external world, generating and controlling displays, file and device I/O and other facilities all of which are well developed and widely available on traditional (UNIX) operating system platforms.

Technical aspects of the mechanism – dubbed α /TMM – for Trivial Machine Monitor, are presented as are details of the OS level RPC facilities providing process/thread level communication between hosted operating systems over a TMM provided transport channel.

Architectural alternatives to the α /TMM approach are discussed including fully robust virtual monitors and contemporary micro-kernel/OS-server designs.

Finally our approach is assessed relative to alternatives and tradeoffs dictated by the particulars of α and our development requirements.

Introduction

The α operating system is a distributed, object-oriented real-time system for mission critical applications. The last of these descriptives means aerospace, military, industrial, scientific and technical high level system command and control. To adequately address these problem domains, unique approaches were taken in α 's design and implementation such that it has little in common with UNIX and UNIX derived OSs nor with conventional real-time systems.

Its development and application however have generated a considerable set of requirements for services and facilities, common on UNIX and conventional real-time operating systems. These include:

- Internet Communications Protocols;
- File and Device I/O;
- X Display Services;
- Hosted Software Services¹.

¹An example of particular interest to us and our research sponsors is CRONUS, a UNIX hosted distributed operating system developed at BBN (see Schantz (1986)). The interface and access issues it presents generalize to other database, communications and computing applications.

The imperatives of developing a new operating system with a finite resource base and a principled aversion to reproducing facilities well-developed and available elsewhere have collectively caused these issues to be deferred and/or dealt with in ad-hoc and unsatisfying ways.

This paper presents details of a relatively simple mechanism for providing convenient, uniform, and low overhead access to these facilities via a UNIX operating system, running on the same computer as α . This mechanism is very straightforward for a narrowly defined but significant set of parallel machines, requiring neither the re-implementation of desired facilities (as native services under α) nor the expansion of α system services (such as would support operation of these facilities in their original form).

This mechanism is a very special case of virtual machine monitors for SMP parallel processors that turns out to be easy to implement and incurs very little operating overhead on the hosted operating systems. We call our implementation of this mechanism α /TMM for Trivial Machine Monitor.

The sections that follow provide, first brief introductory descriptions of α and Concurrent's primary commercial operating system, a real-time enhanced UNIX called RTU, (Real Time UNIX –

under which α system development and applications programming are hosted) and the Concurrent R8000 (MIPS R3000) multiprocessor computers on which both these OSs run.

Following that, the design and implementation of α /TMM is covered, including its machine management and operation support and its in-memory message transport facility for inter-OS communications. The OS level RPC facility implemented on top of the TMM message transport is then discussed, along with the (hosted OS level) UNIX server and α system service object that actually provide UNIX system services to α client threads.

Allowing for a bit of revisionism in our perspective², the TMM is then contrasted with a variety of alternative approaches to solving the general problem of abstracting the implementation of a particular set of services from an operating system into a more manageable construct such that multiple "virtual system environments" (system service suites, operating system servers, etc.), can coexist on a single physical machine.

It is finally argued that while there are serious limitations to the approach taken, that it solves a significant portion of the general problem in a conspicuously simple and effective way such as may be worthy of consideration for applications other than α 's.

The Alpha Operating System

The development of α and the research underlying it have been under way since the early 1980s. α originated at the Department of Computer Science at Carnegie Mellon University. It is currently ongoing at Concurrent Computer in Westford Ma.. We are presently laboring to produce the third major release of the system which is to serve as the base for considerable downstream commercial development of the OS itself and of a range of applications on top of it.

The α operating system addresses an emerging and increasingly challenging problem domain. It does not focus on low-level sampled-data, control loop real-time applications. It deals rather with large, more complex, distributed systems – found first in military warfare environments (in mission support for combat platforms, battle management and C3I), but increasingly now in industrial and aerospace and commercial applications (factory automation, air traffic control, telecommunications, transaction processing). In such applications the focal problems for operating systems are:

- Distributed Processing;

²We didn't set out to solve the general problem, just to do some practical things along the way of α development. However, recast from the perspective of the general problem, it merits presenting in that context.

- Flexible, Policy-Driven Scheduling;
- Fault-Tolerance.

All of which support the construction of high level control systems that can operate in complex, changing (and dangerous) environments – insuring the best possible results (under potentially deteriorated circumstances) and possibly with limited human intervention³.

In α these problems are addressed with:

- Object/thread oriented system interface – A system wide/kernel protected capability space provides execution threads access to transparently network-distributed objects, operation invocations on which are the basic α computing paradigm⁴;
- Time value function driven scheduling – Highly rationalized scheduling of thread execution based on (application programmer) expressed time-value functions which define the utility of completing a given computation as a function of time. This conceptual basis supports a range of scheduling approaches and algorithms that address – in the context of ongoing intensive research on real-time scheduling technology – issues of task dependency, concurrency and parallelism and adaptive fault tolerance⁵;
- A kernel scheduler organization designed as a scheduling testbed, providing a modular scheduler interface under which a range of scheduler implementations – both conventional and experimental are in use, or are under development and test.
- Transparent, distributed, parallel processing supported *within* α nodes by a fully symmetrical shared data α kernel operating each node, and *between* nodes by a special suite of α communications protocols used by the kernel to provide transparent object and thread distribution (including the remote invocation protocol (RI), the thread management and repair protocol (TMAR) and the page transfer protocol (PT, supporting object replication));
- Facilities provided for user parallel programming, including concurrency control primitives and thread/object creation/management

³See Jensen and Northcutt (1990) and Northcutt (1987) for variously detailed overviews of the system.

⁴See Northcutt et al (1990) for details on the α thread-based execution model.

⁵Time-driven resource management, of which scheduling is a fundamental special case, is a central theme in α . The seminal work on time-value function models is due to E. Douglas Jensen, now Concurrent Chief Scientist and leader of the α Research Group. The original presentation is in Jensen 1975. See Jensen, et al (1985) for a more up to date overview. See Clark (1990) and Locke (1986) on current scheduler research and designs.

- operations;
- Sophisticated exception handling mechanisms, support for atomic operations on objects and other features for fault tolerant application development⁶.

All of this is to say that α takes a range of approaches widely divergent from those of traditional UNIX operating systems in an effort to provide new ways of dealing with different and challenging problems. The decision to exclude traditional interfaces and facilities from the specification of the α kernel has contributed to the robustness and power of the facilities that are provided.

Note that there is a high degree of orthogonality between α and UNIX in that there is relatively little overlap of function between the sets of kernel services they provide nor commonality in the ways in which they provide them.

Concurrent RTU

Compared to the above there is relatively little that needs to be said to this audience about RTU. Developed from Bell UNIX releases in the early 1980s, it is currently SVR3.2 compatible. It incorporates a wide range of Concurrent enhancements to support traditional types of real-time programming

- Expedited interrupt handling;
- Real-time priority scheduling;
- Real-time processor, memory and device control (pre-allocation and task dedication);
- User level multiprogramming and synchronization support (threads);
- A variety of Berkeley and other enhancements to facilitate systems programming in general.

From the point of view of kernel particulars affecting the implementation of the TMM and the partitioning of the machine, there is little that distinguishes it from a conventional UNIX kernel.

Host Platform

Both α and RTU are portable operating systems for which support has been developed for a variety of hardware bases including Motorola and MIPS and other, mono and multiprocessor systems. Our current target both for ongoing α development and α /TMM implementation, is a new generation of Concurrent multiprocessor systems based on the MIPS R3000 chips set⁷. These feature:

- 25MHz R3000 cpus with R3010 floating point units, two per processor subsystem with 64k

⁶Design details for the α kernel are given in Northcutt and Clark (1986). The α system (release 2.0) interface is presented in Shipman (1990).

⁷Hardware design details are given in Rungsea (1987) and Solomon (1989).

(per processor) coherent (write through) caches and a (4 word deep) buffered (write) memory interface;

- ECC, buffered memory, 64 bits wide;
- Hardware support for multiprocessor interrupts and synchronization (via a private bus⁸);
- A variety of configurations supporting up to 16 processors and 256MB of memory.

Each processor has private serial ports which variously support console operations and debugging using an α variant of MIPS remote dbx (IDT (1989)). Each processor also has a relative abundance of (private) timer support hardware. The expected range of (VME bus) devices is provided along with RTU system support. Of primary interest to α are the current (AMD LANCE) based Ethernet controller and forthcoming FDDI interface.

The R8000 hardware is representative of a class of small to middle scale parallel processors that are coming into fairly widespread contemporary use. Various details of its implementation (especially the per processor timers and serial ports significantly simplify the TMM design – requiring no work to support sharing them between hosted systems).

TMM Architectural Overview

The opportunity here arises from the joint availability of the various real operating systems in whose services one is interested, and an SMP machine on which they all operate. Relatively little is needed in the way of machine support (the current implementation for MIPS cpus providing an existence proof of the scheme's feasibility for architectures devoid of support for difficult sorts of systems programming⁹) given a straightforward SMP system design – (featuring) absence of memory hierarchy, support for multiprocessor synchronization and symmetry of access to system facilities with respect to each of the cpus such that each can do what we will without disturbing anyone else.

The Trivial Machine Monitor abstracts from the hosted operating systems, a minimal set of functions from the bottom of the machine interface sufficient to support dividing the machine into a virtual pair of systems sharing the real system backplane and devices¹⁰. The TMM is illustrated in relation to the

⁸Of the sort used by Sequent (Beck and Kasten (1985)).

⁹The MIPS 2000/3000 architecture, consistent with its reduced instruction set design, provides minimalist support for operating systems. It does not provide direct support for memory management; it does not provide vectored interrupts nor significant support for state transitions associated with traps and exception handling. For architectural details, see the MIPS processor handbook (Kane (1988)).

¹⁰The TMM implementation currently supports two hosted operating systems. Support is trivially expanded to handle more (but this is unlikely to happen within my attention span).

machine and the hosted operating systems in Figure 1.

The indirect machine interface depicted in the figure corresponds to the (small) set of features taken over by the TMM including:

- Configuration Management;
- System Initialization;
- Interrupt and Exception Handling;
- Lock Management.

The direct machine interface encompasses everything else; the vast majority of the real machine interface and such as:

- Scheduling;
- Execution (process/thread) Management;
- Space (process/object) Management;
- Virtual Memory Management;

The TMM additionally provides a simple multiplexed message transport to facilitate communication between operating systems. It is interrupt driven and requires code at both the TMM and OS levels to operate.

Since the TMM is not a virtual machine monitor (such as would allow one (or each) processor to execute, from time to time, several operating system programs) its active component is very small – not providing any facilities for saving and restoring processor and machine state¹¹. Each partition of the real system (consisting of a configurable number of the available processors and a range of the available memory) executes its assigned operating system essentially as it does when that operating system has

total control of the system.

The MIPS architecture does not provide a practical mechanism for separating the address spaces of the various players nor for implementing relative privilege for monitor and hosted systems – all execute with kernel privilege in the kseg0 (unmapped and cached) address space. I had a strong initial desire to make this work in mapped space (kseg2) but the impact that would have had (and the labour required) on the hosted systems discouraged me from that.

This has had several significant impacts. On the win side perhaps is that the interface between TMM and hosted systems is simple and direct (procedure calls). On the loss side beyond the obvious lack of protection, is that all the system text and data (for both TMM and the operating systems) is not relocatable – addresses, memory allocations and layouts, etc. must be set at system build time.

TMM Implementation

The TMM and hosted operating systems are bootstrapped by the standard rom bootstrap using a scheme (of convenience) in which the hosted operating systems are appended to the data segment of the TMM. A simple bootstrap overloader recursively accomplishes this, filling out the BSS segments to size at each turn¹².

¹²The result is a somewhat overgrown boot file and therefore somewhat delayed bootstrap process. I will eventually crack the COFF executable file header sufficiently to add segments for the hosted operating systems to the TMM header and understand the rom bootstrap sufficiently to coax it into conveying all of them into the appropriate addresses in memory. The current

¹¹Compare Doran (1988) and/or Parmelee (1972) for examples of the “real VMs don’t eat quiche” genre.

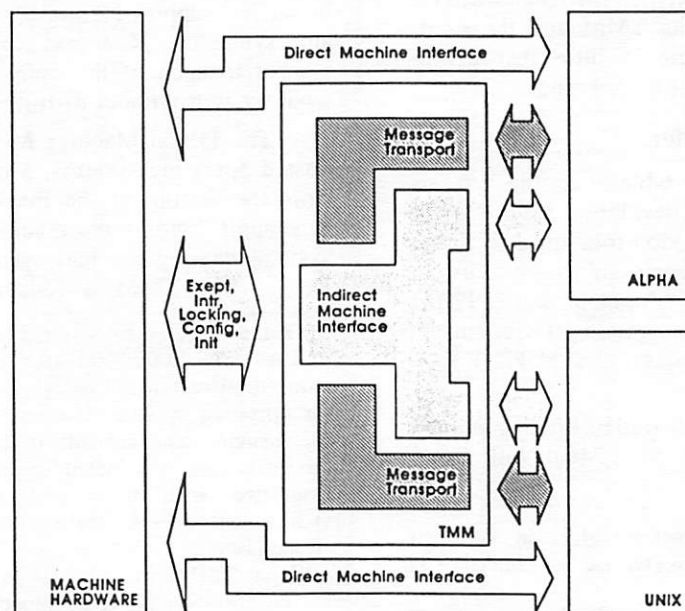


Figure 1: ALPHA/TMM Functional Diagram

The memory layout of the system, processor and device allocations are created at system build time. The TMM and hosted OSs are loaded into zones in low memory (the 0th megabyte for the TMM and message buffers, two megabyte zones above it for each OS). Memory above the system images is partitioned at boot time and allocated to each OS allowing for some flexibility in dealing with the physical configuration of the hardware boot device¹³. Each hosted OS configures and frees into its free page management structures, pages from the end of its kernel data segment to the end of its low memory zone as well as all its allocated high memory pages¹⁴.

High memory, processor and device configuration data as well as entry points for relevant procedures in the TMM, are passed to the hosted OSs as they are started up as pointers to per-OS configuration structures in the TMM data segment¹⁵.

To complete the configuration and initialization story, note that rom bootstrap begins execution at the entry point in the original (TMM) header (at "start" in the TMM). It sizes and allocates memory and counts and allocates processors – thus defining the partitions of the machine (and notes them in the OS configuration structures). It then starts the 0th processor in each partition at the (well known) entry point of the corresponding operating system.

The hosted OS startups were modified to the extent necessary to assimilate TMM provided configuration information before jumping into startups otherwise unchanged from the stand-alone versions.

Interrupt and exception handling turns out to be easy. The MIPS architecture doesn't provide vectored exceptions. Code in the TMM at the fixed machine exception locations identifies the executing

scheme was pirated from a private α mechanism for overloading system initialization and application objects onto the kernel data segment.

¹³Currently the rest of memory is divided in half and half given to each OS. Two processors are allocated to UNIX and the rest to α (our various test beds and development machines each have at least four cpus). Other policies are trivially implemented. At some point soon a configuration generator will be available to optionally patch the TMM load image or R8000 static memory, with system run-time configuration data.

¹⁴Both systems as originally constituted, readily handle discontinuous physical memory, so very little work was required to implement the zone arrangement. A run time facility for occasional memory (and/or processor) stealing (supported by code in the hosted systems and entries in the TMM for initiating and resolving the requests) is easily devised. Aside from that, memory (and processor) allocations are static from boot-time.

¹⁵Hosted OS startup subsequently fills in OS configuration information for the TMM in reserved fields of the same structure.

processor and jumps to an operating system specific handler corresponding to that processor that incurred the fault (received the interrupt). Give or take a small amount of address bashing in the hosted OS exception handling code, it runs as per original (ie. as it would stand-alone).

An implementation for a machine with vectored interrupts and a relocatable vector table (Intel, Motorola and other CISC and RISC cpus) would presumably require simply the correct per-processor initialization of OS specific vector base registers.

The R8000 provides hardware support for synchronization in the form of a special bus and small memory for synchronizers. In the partitioned system, this memory is divided into three ranges, one for private use by each of the hosted operating systems and one for use arbitrated by the TMM to lock shared resources (principally message buffers and control structures (see ahead) and shared physical devices (principally the ethernet LANCE).

Meaningful device sharing in schemes like this typically is non-trivial to achieve. Fortunately in our case, the foci of the hosted operating systems are widely different and this reflects on their use of system peripheral devices. Principally, the real I/O devices (disk, tape, etc.) are left to the exclusive use of the UNIX side¹⁶. The exception is the ethernet interface, the software support for which is partitioned into OS level shell drivers that operate the device through a real driver migrated into the TMM. This runs the LANCE in multicast mode so as imitate two "virtual" network addresses with the one device allowing each OS to act as an independent host on the network¹⁷.

TMM Inter-Host Message Transport

A simple multiplexed in-memory message channel is implemented in the TMM. This provides a reliable message medium between client server pairs spanning the hosted operating systems. The MIPS implementation of this message channel is very straightforward. Since the TMM and the hosted systems all live in the kseg0 space, the actual

¹⁶Efforts to prototype permanent object storage systems for α are currently underway which are generating requirements for a native disk I/O interface (these are using the α /TMM to simulate a backing store with UNIX files) which will eventually need to be satisfied for the TMM using separate drives and or controllers or partitioned single drives supported at a sub-driver level in the TMM.

¹⁷In this way two network communications paths are provided for α as originally intended. The original native protocols continue to function using the direct device interface through the TMM. These are now supplemented with additional (outside world) internet communications provided by the UNIX side (via RPC over the TMM message transport; see following sections).

message buffer memory is permanently mapped into everyone's address space. Largish (1k) buffers are allocated from a free chain to client subchannels. Subchannels are tagged in a message buffer header and hashed by thread/process number (on α and UNIX ends respectively) onto subchannel chains corresponding to each direction of traffic.

Interprocessor interrupts drive the channel. When a message is enqueued on an idle channel, an interprocessor interrupt is posted activate it. Per chain and per buffer inter-OS (shared) locks protect the data buffers and control structures. The implementation provides safe access to the queues while the transport is running such as should eliminate interrupt overhead for message arrival rates above a certain threshold.

Kernel operations in α create/destroy subchannels on demand. The UNIX end is presented as a special file to server daemons. Each open of the file blocks until a connection is established from the α

side¹⁸.

¹⁸The UNIX side service daemon sleeps on the the open of the message channel device, forks a server child process for the client that is connecting when the open completes and then re-opens the device to wait for the next connection. I am open to criticism of the low level at which this interface is cast. It is simple, portable and sufficient for the essentially private communication between hosted systems that it supports. There is fully developed Berkeley IPC on the UNIX side and no intrinsic need for anything like it on the α side (there is of course the extrinsic need for an interface to the former for α that is a stated point of the exercise). Note also that an alternative processor architecture providing hardware support for messaging (with access checking and automation for data transfers) might have made for a more robust implementation (one which kept everyone cleanly in their own address space). It would certainly have been more difficult to do.

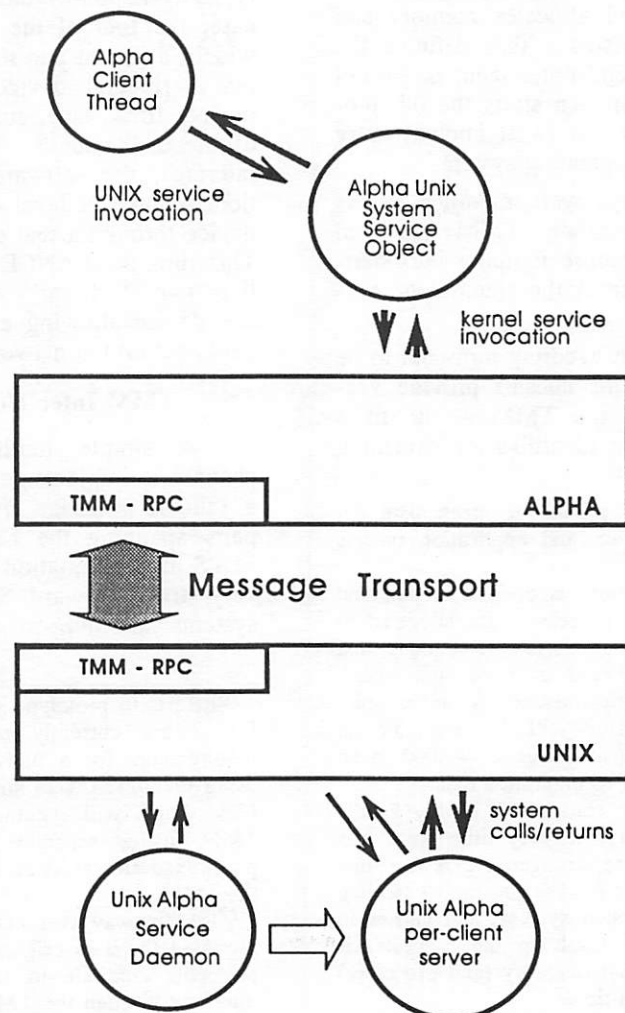


Figure 2: Alpha - UNIX RPC Communications Path

TMM Hosted Operating System Communications

Operating system to operating system communications are implemented at the hosted operating system level with a special RPC designed for this purpose that runs over the TMM provided message transport. Figure 2 illustrates the communications path from end to end – from α client thread to UNIX server process and back¹⁹.

The α client thread invokes “Alpha UNIX System Service Object” operations, examples of which are given in Figure 3. The server object operations call per-service RPC stubs that generate messages over the TMM message transport to a UNIX side system call server. This issues UNIX system calls on behalf of the client and returns to it the results²⁰.

In designing this we looked fairly closely at Sun RPC (Sun Microsystems (1988)) in an effort to avoid building a new one. We elected to do so anyway for three reasons:

- Sun’s RPC is very intimately tied to the Berkeley IPC communications model for which we have no support on the α side;
- I thought it would reduce the implementation

¹⁹The α /TMM implementation provides for the moment, UNIX services to α threads only. Service in the other direction can and may be added with incremental work as the need arises. Existing facilities should generally support it without or with trivial modification.

²⁰See Stevens (1990) for an overview of RPC mechanism basics.

cost of the service layer if we had more flexible parameter/return handling than provided by Sun (which supports a single argument and single return only and thus requires one to package more elaborate constructs manually²¹);

- Sun’s RPC is also intimately tied to XDR (Sun Microsystems (1987)). Operation on heterogeneous networks is an issue which we are attempting to address in a systematic way for α such that the data representation is managed uniformly from end to end of the kernel implementation of invocation. We are therefore trying to take a systematic look at data representation schemes (including ASN, XDR and others) with an eye to solving this problem at a level above the α /TMM RPC interface²².

²¹This likely has a performance impact as well from passing more data with each procedure call than one otherwise would.

²²For the moment we pass raw data between hosted operating systems. This works perfectly well since all are running on the same type of cpu and use the same C language toolset. As of this writing an experiment with this RPC using CASN.1 – the ASN.1-C compiler by Neufeld and Yang (1990) is also underway, to gain some familiarity with that toolset and develop impressions of the costs involved (in implementation and performance). Note that the case of heterogeneous processors on the same backplane is an especially interesting one, such configurations having been specified by the Navy for its Next Generation Computer Resource development (NGCR

```
/*
 * Alpha UNIX Service Object -- Object operation
 * interface around inter-OS rpc calls to UNIX
 * side system call server. Notice errno gets
 * returned as an extra parameter. Omitted system
 * calls are not supported (feel free if you
 * want one -- nVa 10.11.90).
 */
OBJECT UNIX_Call {
    int errno;
    :
    :
    /*
     * 3 -- read
     */
    OPERATION read(IN int fd,
        IN int size,
        OUT int count,
        OUT char buf[MAX_READ],
        OUT int errno ) {
        char buf[MAX_READ];
        count = ux_srv_read(fd, &buf,
            size, &errno);
    }
    :
    :
}
```

```
/*
 * 4 -- write
 */
OPERATION write(IN int fd,
    IN int size,
    IN char buf[MAX_READ],
    OUT int count,
    OUT int errno) {
    count = ux_srv_write(fd, &buf,
        size, &errno);
}
/*
 * 98 -- connect
 */
OPERATION connect(IN int s,
    IN struct sockaddr name,
    IN int namelen,
    OUT int rval,
    OUT int errno) {
    rval = ux_srv_connect(s, &name,
        namelen, &errno);
}
:
:
```

Figure 3: Alpha – UNIX Service Interface

The α /TMM RPC implementation provides a simple table driven stub generator which translates procedure, host, server and version identifications, parameter descriptions and parameter/return handling specifications into calls on a small library that perform the usual sequence of operations on client and server sides. Especially useful for dealing with the complexities of the UNIX interface are parameter/return handling provided including:

- Shadowed by reference parameters;
- Conditional parameter transmission;
- Conditional return transmission;
- Shadow return values.

The stub generator could do with a more lexically oriented front end but the code table interface works acceptably. We have yet to make any serious measurements of our RPC implementation's performance. While there is clearly room for improvement in the design, it seems to be meeting expectations in the limited use it has seen so far²³.

Note at last that the UNIX service operations on the α side are still a very low level interface from a real work in α point of view. Intermediate objects that use these operations to provide higher level services (an α object store for example) intervene adding arbitrary further levels of abstraction to the interface.

Discussion

The α /TMM is reasonably contrasted to several classes of systems which provide multiple abstracted operating system personae to the user.

Robust (mainframe) virtual machine monitors (of the IBM and Amdahl type cited above) is the first class of things that comes to mind. These do a lot of hard work that is not within the scope of the TMM. They share single cpus among hosted operating systems, transparently preserving and restoring machine state across transitions and often providing different virtual views (tailored to each hosted OS) of the underlying machine as well.

The TMM approach taken here is substantially more minimalist and requires both significant support from and assumptions about the underlying machine as well as cooperation from (meaning accommodating changes in) the hosted operating systems.

²³ – see U.S. Navy (1990)).

²³ Performance measurements are on my personal agenda such as would facilitate a more rigorous (quantitative) assessment of the direct cost of running an application with a dual OS, in the manner described. There is considerable room for improvement in our admittedly crude RPC implementation. The shared-memory and private-communications-only model under which it was designed urges a range of (straightforward in the model) lazy copy and mapping optimizations.

Contemporary operating systems with hierarchically organized kernels and services, such as Mach (Accetta et al (1986) and Golub et al (1990) on the particular point here), Chorus (Rozier et al (1988)) and Taos (Thacker et al (1988), the DEC Firefly operating system) seek to solve, from the same general motivation as α /TMM the more general problem of providing a level of abstraction within the context of the operating system itself for implementing contrasting and or complementing system service suites.

The history of UNIX development over the last half decade reveals considerable architectural innovation coming out of attempts to stretch and extend the programming models and services provided by traditional UNIX implementations. The history of α development over the same period contrasts this reflecting a conscious preference for redesign and reimplementing from scratch. Nonetheless we too have identified in the end a requirement for extending our new system interface to include foreign services – in this case the traditional UNIX base.

However that may be there are a few comparative points that may be worth trying to making:

- The micro-kernel and message kernel architectures have a generality, and architectural richness that the TMM monitor lacks; this implies extensibility and flexibility that promise interesting future developments;
- They are expensive to implement – the kernel level is limited in scope and facilities but the system servers that run on them are in varying degrees new domains to be explored;
- There are serious performance problems to be addressed – all the messaging underlying system service invocation and inter-server communication is not free;
- The trivial monitor is by contrast, not very general – providing only specific (RPC) interfaces between cooperating pre-existing operating systems;
- It does however provide significant enrichment to application development within the hosted operating systems in the form of relatively painless access to the facilities of the other resident OS.

Conclusions

The α /TMM is an opportunistic exploitation of circumstances that appears to work rather well:

- Minimal code is required in the TMM to support some indirection of the machine interface and inter-OS communications;
- Minimal changes are required at the hosted operating system level to play with the TMM.

The TMM is also efficient and effective:

- The stipulated SMP machine architecture supports partitioned operation well.
- Little to no overhead (in proportion to the extent of inter-operating system communications traffic) is added to the hosted operating systems²⁴;
- The partitioned system hosting multiple operating systems provides a substantially enriched environment – giving thread/process level access to two OS service suites at the same time.

There are nonetheless, significant limitations:

- The view from the application programmer's perspective is far from seamless and consistent. Aspects of the foreign system logically clash with the local system. Corresponding services are simply ignored without attempt to resolve the logical problems.
- Translations from the logical space of one operating system to that of the other are accomplished only via real work at the system/applications development level²⁵;

Anyway, the TMM is not, nor was it ever intended to be, the ultimate solution to the problem of abstracting and extending the α system interface²⁶. However it serves the purpose it was for which it was created reasonably well and was surprisingly easy to implement. I therefore recommend the approach in cases where the machine supports it and there is a need to bridge two OS environments for practical application building.

Acknowledgements

Please note that α research and development at Concurrent and elsewhere, is sponsored in part by the U.S.A.F. Rome Air Development Center (RADC); additional support has been provided by The U.S. Naval Ocean Systems Center and the General Dynamics, IBM and Sun Microsystems corporations.

²⁴Considered in contrast to micro-kernel and messaging kernel systems, where the basic nucleus is message oriented and messaging overhead is incurred on every system service invocation, the economy of the TMM approach is significant.

²⁵In our case with intermediate objects layered between actual α clients and the UNIX server object.

²⁶The α kernel object model provides an extensible native facility for expanding the kernel service base. A native implementation of POSIX services is under consideration to satisfy developing interest in our user community. This work however provides an effective testbed on which to experiment with how such an implementation might be structured and thus to insights as to what the tradeoffs inherent in various approaches might be.

The author wishes to thank his colleagues in the α Research Group for their input to the α /TMM design and this paper, especially Ray Clark and Sam Shipman for valuable criticism of his eleventh hour draft and to thank Rob and the USENIX Typography Mavens for their indulgence of his dog's disreputable personal habits.

References

- Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the Summer 1986 USENIX Association Technical Conference*, Atlanta GA.; pp. 93-112.
- Bob Beck and Bob Kasten, "VLSI Assist in Building a Multiprocessor UNIX System", *Proceedings of the Summer 1985 USENIX Association Technical Conference*, Portland Oregon; pp 255-275.
- Raymond K. Clark, *Scheduling Dependent Real-Time Activities*, Carnegie Mellon University, Department of Computer Science, Ph.D. Dissertation, August 1990.
- Robert W. Doran, "Amdahl Multiple-Domain Architecture", *IEEE Computer*, Vol. 21, No. 10, October 1988; pp. 20-28.
- David Golub, Randall Dean, Alessandro Forin and Richard Rashid, "Unix as an Application Program", *Proceedings of The Summer 1990 USENIX Association Technical Conference*, Anaheim, CA.; pp. 87-95.
- Integrated Device Technology (IDT), *IDT RISC : R3000 Family System Programmer's Package Reference*, IDT, 1989.
- E. Douglas Jensen, *Time-Value Functions for BMD Radar Scheduling*, Technical Report, Honeywell Systems and Research Center, June 1975.
- E. Douglas Jensen, C.D. Locke and Hideyuki Tokuda, "A Time Value Driven Scheduling Model for Real-Time Operating Systems", *Proceeding of the 1985 IEEE Symposium on Real-Time Systems*, IEEE; November, 1985.
- E. Douglas Jensen and J. Duane Northcutt, "Alpha: A Non-Proprietary Operating System for Mission-Critical Real-Time Distributed Systems", *Proceedings of the 1990 IEEE Workshop on Experimental Distributed Systems*, Huntsville, AL., October 1990.
- Gerry Kane, *MIPS RISC Architecture*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1988.

- C.D. Locke, *Best-Effort Decision Making for Real-Time Scheduling*, Carnegie Mellon University, School of Computer Science, Ph.D. Dissertation, June 1986.
- U.S. Navy – Next Generation Computer Resources (NGCR) Operating Systems Working Group (OSWG), *Evaluation Results Report for NGCR Operating Systems Interface Baseline Selection*, U.S. Navy Technical Report, May 1990.
- Gerald W. Neufeld and Yueli Yang "The Design and Implementation of an ASN.1-C Compiler", *IEEE Transactions on Software Engineering*, Vol. 16, No. 10; October 1990.
- J. Duane Northcutt, *Mechanisms for Reliable Distributed Real-Time Operating Systems: The Alpha Kernel* Academic Press, Orlando Florida, 1987.
- J. Duane Northcutt and Raymond K. Clark *The Alpha Operating System: Kernel Internals – Archons Project Technical Report 88051* Carnegie Mellon University, School of Computer Science, May 1986.
- J. Duane Northcutt, Raymond K. Clark, Samuel E. Shipman, David P. Maynard, E. Douglas Jensen, Franklin D. Reynolds and B. Dasarathy, "Threads: A Programming Construct for Reliable Real-Time Distributed Programming", *Proceedings of The International Conference on Parallel and Distributed Computing and Systems*, International Society for Mini and Micro Computers, October, 1990.
- R.P. Parmelee, T.I. Peterson, C.C. Tillman and D.J. Hatfield, "Virtual Storage and Virtual Machine Concepts", *IBM Systems Journal*, Vol. 11, No. 2, April 1972; pp. 99-130.
- Marc Rozier, Vadim Abrossimov, Francois Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frederic Herrmann, Claude Gaiser, Sylvain Langlois, Pierre Leonard and Will Neuhauser, "CHORUS Distributed Operating Systems", *Computing Systems Journal*, Vol. 1, No. 4, The USENIX Association, December 1988, pp. 305-370.
- Mana Rungsea and Allen Yee, *Sync Bus Controller Chip (Functional Specification)*, Silicon Graphics Corporation, November 16, 1987.
- R.E. Schantz, R.H. Thomas, and G. Bono, "The Architecture of The Cronus Distributed Operating System", *Proceedings of The Sixth International Conference on Distributed Computing Systems*, IEEE, 1986.
- Samuel Shipman, *Alpha Operating System, Release 2.0 – Kernel Interface Draft (Revision 0.3)*, Concurrent Computer Corporation, August 1990.
- Doug Solomon, *IP7 Functional Description*, Silicon Graphics Corporation, February, 1989.
- W. Richard Stevens, *Unix Network Programming* Prentice-Hall Inc., Englewood Cliffs, N.J., 1990.
- Sun Microsystems, *XDR: External Data Representation Standard*, RFC 1014; June 1987.
- Sun Microsystems, *RPC: Remote Procedure Call Protocol Specification, Version 2*, RFC 1057; June 1988.
- C.P. Thacker, L.C. Steward and E.H. Satterthwaite Jr., "Firefly: A Multiprocessor Workstation", *IEEE Transactions on Computers*, Vol. 37, No. 8, August 1988; pp. 909-920.

Nick Vasilatos is a Consulting Engineer in the α Research Group at Concurrent Computer Corporation. He holds a B.S. degree from the Massachusetts Institute of Technology. His research interests include multiprocessor, multi-computer and distributed system architectures and operating systems, especially for real-time computing.



He is boxer@westford.ccur.com on the Internet. He can be reached via U.S. Mail at Concurrent Computer Corp., One Technology Way, Westford Ma. 01886.

Extent-like Performance from a UNIX File System

L. W. McVoy, S. R. Kleiman – Sun Microsystems, Inc.

ABSTRACT

In an effort to meet the increasing throughput demands on the SunOS file system made both by applications and higher performance hardware, several optimization paths were examined. The principal constraints were that the on-disk file system format remain the same and that whatever changes were necessary not be user-visible. The solution arrived at was to approximate the behavior of extent based file systems by grouping I/O operations into clusters instead of dealing in individual blocks. A single clustered I/O may take the place of 15–30 block I/Os, resulting in a factor of two increased sequential performance increase. The changes described were restricted to a small portion of the file system code; no user-visible changes were necessary and the on-disk format was not altered.

Introduction

File systems are a common place to find performance problems. The original UNIX file system [Thompson] is elegant in its simplicity: it has a single block size and a simple list based allocation policy. [McKusick] describes the drawbacks of this design and also describes Berkeley's fast file system (FFS). The fast file system solves many performance problems found in the original UNIX file system. The fast file system is the basis for UFS, Sun's UNIX File System.¹

UFS has served us well for several years. However, both applications and disk subsystems are demanding higher and higher transfer rates through the file system. Applications such as video and sound require much higher data rates than are available today through UFS. Disk subsystems, such as disk arrays [Patterson], are being developed to deliver the desired I/O rates. Measuring the existing UFS showed that about half of a 12MIPS CPU was used to get half of the disk bandwidth of a 1.5MB/second disk.

Goals and constraints

It was clear that the current implementation of UFS did not scale to the desired I/O rates, so we set out to improve the system. We wanted a UFS that used less CPU to run the disks at their full bandwidth. An additional goal was that *all* users of the file system should benefit from the enhancements; the primary constraint was that the on-disk format of the file system could not change. The "dusty-deck" approach insured that no application would need to be aware of the enhancements.

¹UFS has been modified to fit into Sun's virtual file system architecture [Kleiman]. Other than that, it has been tracking the fast file system very closely.

This paper describes an enhancement to UFS that met all our goals. The remainder of the paper is divided into seven sections. The first section reviews the relevant background material. The second section discusses several possible solutions to the performance problems found in UFS. The third section describes the implementation of the solution we chose: file system I/O clustering. The fourth section discusses problems found in the interaction between the file system and the VM systems. The next section presents performance measurements of the modified file system. The sixth section compares this work to other work in this area. The final section discusses possible future enhancements.

Background

To understand our UFS enhancements, it is necessary to understand the basics of the SunOS Virtual Memory (VM) and Virtual File System (VFS) architectures². A brief review is presented here. More details on the VM system may be found in [Gingell] and [Moran]. Readers familiar with the interaction between the VM system and a file system, in particular the `rdwr`, `getpage`, and `putpage` VFS interfaces, may wish to skip forward to the section on UFS performance problems. Readers familiar with either FFS or UFS, in particular the reasons for its rotational delay, may skip past the section on UFS performance problems. Readers are expected to understand the original UNIX I/O system (the buffer cache) explained in [Bach] and [Ritchie].

²The VM and VFS architectures are similar to those in System V release 4. Virtually all references to SunOS are also applicable to SVR4.

Virtual file system interfaces

The SunOS virtual file system (VFS) interface [Kleiman] allows the kernel to support many different types of file systems simultaneously. Each file system type implements two object classes: *vfs* and *vnode*. A VFS object represents a particular instance of a file system. A vnode object represents a particular file within a VFS. These objects export interface routines that the main body of the kernel uses to manipulate a file system without knowing the details of how it is implemented. A file system type may be thought of as a driver that provides a set of file system abstractions without exposing the details of the implementation.

There are many entry points into a VFS, but we need concern ourselves only with the read/write (*rdwr*), read a page (*getpage*), and the write a page (*putpage*) interfaces. These are the interfaces used by the *read*, *write*, and *mmap* system calls that the programmer sees.

The *getpage* interface returns a page filled with data from the vnode at the file offset specified by the caller. The file system may use a page cache supplied by the VM system to store active page data. The entries in this cache are named by the vnode and file offset of the data in the page. The *putpage* interface is used to return a page to secondary storage.

In most SunOS file systems, the *getpage* and *putpage* routines are where the I/O actually occurs. It is important to understand that *getpage* and *putpage* are used asymmetrically. *getpage* is usually called first both for reading and writing. In the read case it is called to retrieve that data from the disk. In the write case it is called to get a copy of the data to be modified. *putpage* is only called when the page is to be written to the backing storage.

When a process uses the *read* or *write* system call, the kernel redirects the call to the *rdwr* entry point of the appropriate VFS. *rdwr* copies the appropriate file data to or from a buffer supplied by the caller. Usually this is the buffer specified by the process in the *read* or *write* system call. Many file systems implement *rdwr* by mapping a portion of the file into the kernel's address space and then copying to or from the user's buffer.

SunOS virtual memory system

The SunOS VM model is similar to that of Multics [Organick] and TENEX [Bobrow]. The VM system works in concert with the file systems to manage a cache of vnode pages. To illustrate the caching mechanism, we describe the VM system's management of a simple address space. The address space, associated with a process, is made up of a collection of segments each of which refers to a portion of a file (vnode).

Figure 1 shows a simple address space made up of two files: *a.out*, a file from a local UFS file system, and *libc.so*, a dynamically linked shared library from a remote NFS file system.

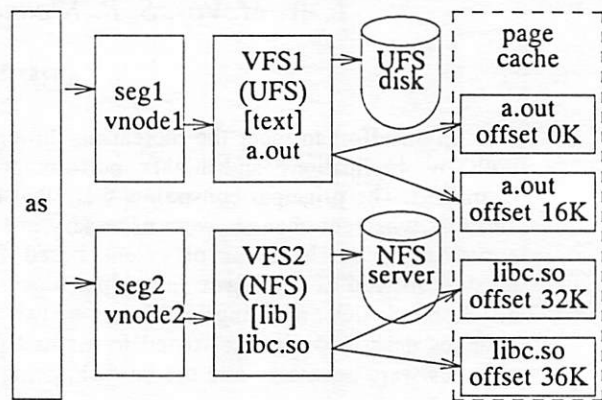


Figure 1: The VM system

Page faults

When a process references an address for the first time, a page fault occurs. The fault is resolved by traversing the object hierarchy and invoking the fault handlers for each object type. Specifically, the kernel finds the address space associated with the process and calls the address fault handler, passing it the faulting address. The address fault handler uses the address to find the enclosing segment and calls that segment's fault handler. The segment's fault handler converts the address into a *<vnode, offset>* pair and calls *getpage* of the associated file system. The *getpage* routine first requests the VM system to find the page denoted by the *<vnode, offset>* argument. If the page is found in the page cache, it is returned. Otherwise, the page is not in memory and the file system has to retrieve the page from secondary store. After the data has been retrieved, the file system puts the page in the page cache for future reference.

An important point is that there is no longer a distinction between process pages and I/O pages. Pages are brought into the system for different reasons but they are all labeled in the same way. This unified naming scheme allows all of memory to be used for any purpose, based on demand. All of memory may be an I/O cache if the system is acting primarily as an I/O server, or all of memory may be used up for a single large active process. Older UNIX variants confined I/O pages to a small "buffer cache."

UFS details

The UFS implementation uses several internal concepts, such as *inode*, *dinode*, *logical block*, and *physical block*. These are explained in [Leffler] but we briefly review them here.

UFS represents each active file with an *inode*. An inode is an in-memory version of the control information associated with a file; the inode is initialized when the file is first read from disk from an on-disk structure called the *dinode*. The inode contains information such as file size, the location of the first few data blocks on disk, date created, etc. Each inode is directly associated with a *vnode*. Inodes also contain meta information that the file system uses to help tune performance. We discuss this information in the *ufs_getpage* section below.

UFS breaks up each file into *logical blocks*. A logical block is the main unit of allocation in UFS³. Logical block numbers, or *lbns*, are numbered from zero and denote a particular block of a particular file. Logical blocks are used for two reasons: to decouple the file system block size from the disk block (or sector) size, and to decouple the location of a block in a file from the location of the block on the disk.

ufs_rdwr

ufs_rdwr performs a read by breaking the request into block sized pieces, mapping each file block in turn to an unused portion of the kernel's address space, copying the data to the requesting process, and unmapping the block.

If the page representing the block is not already in memory with an active MMU translation, the copy will fault. The kernel handles the fault by calling *ufs_getpage* to find the page. After the page is retrieved, the MMU translation to the page is set up, the fault returns, and *ufs_rdwr* finishes the copy unaware that the fault ever occurred.

Repeated accesses to the same page will find the page still in memory with an active translation and will avoid multiple page faults.

The work done for a write is similar. The main difference is that when the block is unmapped from the kernel's address space after each block is copied, *ufs_putpage* will be called to start the I/O to the disk. *ufs_rdwr* can also request that *ufs_putpage* wait until the I/O is complete (synchronous write) or that it return after the I/O has been started.

ufs_getpage

When *ufs_getpage* is called, it first checks to see whether the page is actually already in the page cache and returns the page if it is. Otherwise, it converts the *vnode* and offset into the equivalent inode and logical block number and calls *bmap*, which is responsible for mapping logical blocks of

an inode to physical blocks on the disk as well as the allocation of physical blocks on disk. It uses the block pointers in the inode to perform the translation, unless the file is large, in which case the inode contains a pointer to a disk block of pointers; this block is called an indirect block. For large files, *bmap* needs to fetch the indirect block to perform the translation. The physical block number returned by *bmap* is used to start up the I/O.

The *ufs_getpage* routine is complicated by the heuristics for optimizing read performance. The algorithm is shown in figure 2.

```
bmap() to find disk location
if (requested page not in cache) {
    start I/O for requested
}
if (sequential I/O) {
    do another bmap() if necessary
    start I/O for next page
}
if (first page was not in cache) {
    wait for I/O to finish
}
predict next I/O location
```

Figure 2: UFS getpage algorithm

In the absence of other information, *ufs_getpage* uses the pattern of logical block requests it sees to predict the file access pattern in the near future. If the pattern of requests is such that the current request is one page greater than the last request, it is assumed that the file is being accessed sequentially. If sequential access is detected, *ufs_getpage* predicts that the next access will be to the page following the requested page. In this event, *ufs_getpage* will *read ahead*, i.e., will start the I/O for the page following the one requested.

page 0	page 1	page 2
sync read page 0		
async read page 1	async read page 2	async read page 3
next = 1	next = 2	next = 3

Figure 3: access pattern showing read ahead

The series of events that will cause read ahead is illustrated in figure 3. Each box represents a page and shows what happens when a fault is taken for that page. The first fault (for page 0) will start an I/O read for page 0 and also start up an I/O read ahead on page 1. The next fault (for page 1) will find page 1 in memory and will start up a read on page 2 and so on.

In figure 3, the first page fault caused both the primary read and the read ahead. Since the fault was for the beginning of the file, it may seem that the read ahead heuristic should not have been enabled. The file system uses an inode field, *next*, to predict the location of the next read.

³For the purposes of this discussion, we will assume that the size of a block is always greater than or equal to the size of a page.

When the inode is initialized, `next` is set to zero, predicting that the first read will be the first block of the file. Starting read ahead at the beginning of the file turns out to be a beneficial heuristic.

ufs_putpage

When the kernel wishes to free some pages that contain modified data, it calls the appropriate file system's `putpage` routine. `putpage` simply writes out the page data to the correct location on secondary storage.

UFS performance problems

This section considers the reasons that file system operations in UFS are so expensive. The answer comes in two parts: computational overhead and placement policy. There is little that can be done that will reduce the computational overhead. The computational cost can be amortized by moving more data for each traversal of the file system code. This idea was a basic motivation for the FFS changes to the original UNIX file system. Placement policy is more interesting. Even if we reduced the computational overhead to zero, the file system could not deliver the data faster than half the disk transfer rate.

Placement policy

While UFS has many tuning parameters, including ones that affect the placement policy, it is almost always tuned in the same way.

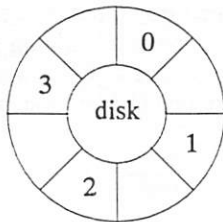


Figure 4: Interleaved blocks

Blocks from a single file are placed as shown in figure 4 in which you are looking down on one track of a disk platter. (The unlabeled blocks will be used by a different file.) The file system is responsible for placing the logical blocks on the disk in a pattern that is optimal for sequential access. Each block is separated by a gap called the rotational delay or `rotdelay` by the file system code⁴. `rotdelay` is specified in milliseconds and the minimum non-zero value is the rotational delay of one block time. For a file system with a block size of 8KB this is 4 milliseconds on typical disks. The number of blocks placed contiguously between each rotational delay is known as `maxcontig`. `maxcontig` is typically set to 1 as shown in figure 4.

Rotational delay

Why is the rotational delay necessary? We already know that the file system does read ahead to avoid delays in sequential access. The rotational delays allow the file system enough time to deliver the current block to the requesting process, for the process to compute using the new data then generate a request for the next block, and for the file system to check that the requested block is in memory (due to read ahead) and generate the disk I/O for the next read ahead block. If the file system is properly tuned, the I/O request will get to the disk as the appropriate block is moving under the head. If there were no rotational delay, the next block would already have started under the disk head by the time the disk saw the request. The disk would have to wait almost a full rotation (about 16 milliseconds on today's disks) before starting that request.

This explains why the rotational delay is necessary but we can see that it comes at a cost: having those holes reduces the maximum transfer rate to half that of the disk rate. To solve this performance problem, the rotational delays must be eliminated and the computational overhead of the system must be reduced.

Possible Improvements

In this section we explore the full range of improvements, from hacks to completely new file system implementations. We reject them all except clustering; the discussion of the extent based file system solution is of special interest.

Raw disk

Get rid of the file system altogether by using the raw disk. Some users, mostly those running database applications, actually do this. There is no question of file system overhead; the raw disk is a direct interface plus a few permission checks.

This solution is an act of desperation. There is no file system, no file abstraction, no read ahead, no caching, in short, none of the features that are expected of a file system. The fact that users resort to the raw disk is usually an indication that the file system is too slow.

File system tuning

Tune the file system to take advantage of track buffers. A track buffer is a memory cache the size of one track commonly found on newer disks, such as SCSI disks, that have on board controllers. When a read request for a block is sent to the disk, the

⁴Note that UFS does this differently than file systems in other operating systems in that the gap is maintained by software. Other systems format the disk to have this gap and call it the disk *interleave*.

entire track is read into the buffer. If successive blocks are on the same track, they are serviced immediately from the track buffer. Therefore, there is no need for rotational delay between successive file blocks. UFS can be tuned to attempt to place successive blocks contiguously on the disk by setting `rotdelay` to zero (see figure 5). This increases read performance substantially, since an entire track's worth of file data can be read in one rotation.

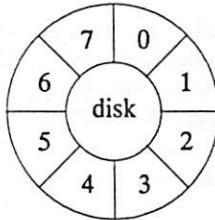


Figure 5: Non-interleaved blocks

At first glance this looks like a win. If we had no rotational delays then a track would contain twice as much relevant data and the effective disk bandwidth would be twice as great. However, not all drives have track buffers. Drives without track buffers would suffer substantial performance penalties on both reads and writes. Still, many of the drives sold today do have track buffers, so why not take the easy way out? The answer is write performance; it suffers horribly when the file system has no rotational delay. The reason for this is that the track buffer acts as write through cache, each write goes through the track buffer to the disk⁵. Since the writes go directly to the disk, we need the rotational delay between each block or each write will wait a full rotation before beginning. Given that writes will degrade and only some reads will improve, we rejected this approach.

Driver clustering

First tune UFS to allocate sequential logical file blocks contiguously by setting `rotdelay` to zero. Then have the disk driver combine (cluster) any contiguous requests in its queue into one large request. This is relatively simple to implement, since many SunOS disk drivers call a routine, `disksort`, that orders the disk queue for optimal seek performance. These drivers call `disksort` each time a new block request is received. `disksort` could coalesce multiple adjacent blocks into one I/O request.

One disadvantage of this approach is that the file system code must be traversed for each block. We felt this to be excessively expensive in CPU

⁵If the block went into the buffer, but not on the disk, the system and/or user may believe that the data is safely on stable storage. If the system crashes the data is lost, even though a promise was made that the data was safe.

cycles. Another problem is that driver clustering helps only writes. The reason for this is that there can be many related writes in the disk queue at once, since writes are asynchronous in nature. Reads, on the other hand, are synchronous, so there can be at most two, the primary block and the read ahead block, in the queue at once. Finally, not all drivers call `disksort`. Instead, those drivers depend on intelligent controllers to do the ordering of requests.

Extent based file system

Replace UFS with a new file system type, an extent based file system. This is a popular answer to file system performance issues. The basic idea is to allocate file data in large, physically contiguous chunks, called extents. Most I/O is done in units of an extent. This improves performance in both I/O rate and CPU utilization, since the I/O is done contiguously, and file system CPU overhead is amortized over larger I/Os. Typically, the user can control the size of these extents on a per-file basis. In most cases the on-disk file system represents the mapping of logical file blocks to physical blocks as a tuple of <logical block number, physical block number, length>. In addition, the on-disk inode is usually expanded to maintain the user's requested extent size(s).

The disadvantage of exposing extents to the user is that it is unlikely that a user will be able to choose the "right" extent size. Even if a good extent size can be determined for a particular file, the size will vary between machines with different configurations, between file systems on the same machine, or even between different locations on the same file system. For example, consider a variable geometry drive (a drive that has more blocks on the outer tracks than on the inner tracks). Such a drive may have different values for the optimal extent size at different locations. The same sort of problem exists when considering a single drive versus a disk array [Patterson]. Trying to write portable code that knows about extents is close to impossible.

Exposing this sort of information to the application is rarely helpful and is frequently confusing. Users rarely want to manage extents. Usually, they really want some sort of performance promise. If the file system performed satisfactorily, the user would never consider telling the file system what to do. We believe that the file system is capable of the required performance with no assistance from the user.

Another disadvantage of this approach is that a change in on-disk file system format would require changes to many system utilities, such as `dump`, `restore`, and `fsck`.

File system clustering

Modify UFS to combine blocks adjacent to the requested blocks into a larger I/O request. This produces most, if not all, of the advantages of an extent-based file system without requiring changes to the on-disk format of UFS.

Clustering Implementation in UFS

This section presents the implementation of the solution we chose, clustering in the file system. The goal of our solution is to realize the full potential of the disk but to incur less CPU cost per byte doing so.

To reach our goal we made two basic changes: we tuned the file system to allocate files contiguously and we changed the file system to transfer sequential I/O in units of clusters. A *cluster* is simply a number of blocks, usually about 56KB worth⁶. This approach solves both of the problems in the old system: the rotational delays are removed, which potentially allows a single file to be read or written at the disk speed, and clusters are used in place of blocks which causes the file system code (and the driver code below it) to be traversed far less frequently than in the old system. The details of our implementation follow.

Allocator details

There were no changes to the allocator. The UFS allocator has always been able to allocate files contiguously. This is almost true; in reality the allocator *tries* to allocate files as requested, but it may not be able to do so if the disk is fragmented. Since our work depends heavily on contiguous allocation, it is important to have confidence in the allocator's ability to allocate contiguously.

Most extent based file systems have the ability to preallocate extents to insure maximum transfer rates. We had originally considered preallocation as well but experience showed that this was largely unnecessary. We tried several tests, ranging from filling up an entire partition with one file to filling up the last 15% of a heavily fragmented /home (users' home directories) partition. In the best case, the average extent⁷ size was 1.5MB in a 13MB file. In the worst case, the average extent size was 62KB in a 16MB file. We expected the allocator to do well when there were no other competing files, but were worried about the fragmented file system case. The results showed us that the allocator thinks ahead enough that it has a good chance of being able to allocate blocks in the desired location. The reason

⁶56KB is used because there are still drivers out there with 16 bit limitations.

⁷Extent is used here to indicate a span of contiguous blocks followed by a gap (unrelated block). An extent may contain any number of clusters.

that the allocator is able to do so well is that it keeps a percentage of the disk (usually 10%) free at all times. The free space is not in a fixed location; the allocator may use any free block at any time as long as it keeps a certain percentage free. It uses this flexibility to do better allocation, good enough that we decided not to "fix" the system by adding preallocation code.

Sizing clusters

We use `maxcontig` to indicate the desired cluster size⁸. Although we ask the allocator to create clusters of size `maxcontig` blocks, the actual cluster size may be less than that. For example, we may want to transfer a 40KB cluster but the portion of the file that we want may be in two 20KB extents on the disk. Somehow, the file system needs to be told that 20KB is the best that can be done at the moment.

The `bmap` routine is able to give us this information since its job is to know about the location of the file on disk. `bmap` used to take a logical block number and return a physical block number. We modified it to return a length as well as the physical block number. The portion of the file starting at the logical block given to `bmap` is located at the physical block returned and continues for at least the length returned. The length returned is at most `maxcontig` blocks long and is used as the effective cluster size by the caller (`ufs_getpage` or `ufs_putpage`).

Read clustering implementation

The implementation of read clustering is in `ufs_getpage`, no changes were required anywhere else (but see the section on page thrashing below). The `ufs_getpage` code still implements the same ideas: do a transfer, predict the location of the next transfer, and if the prediction comes true start the read ahead. The changes in `ufs_getpage` all stem from the switch to clusters from blocks: the rest of the code did not need to be changed. The read ahead implementation, shown in figure 6, is a little different, since we don't do a read ahead on each page, just on each cluster.

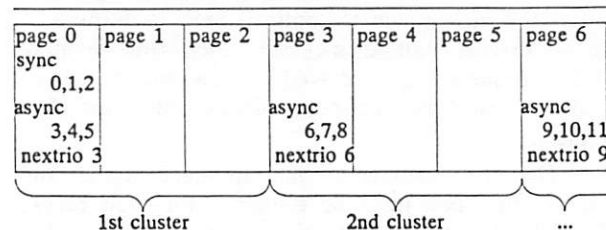


Figure 6 - Clustered reads when `maxcontig` = 3.

⁸Previously, when `rotdelay` was zero, `maxcontig` had no meaning, but now it always indicates cluster size.

As before, each box represents a page and contains the actions that occur as a result of the call to `ufs_getpage` for that page. The first box shows the synchronous read of the first cluster, and the asynchronous read of the second cluster. It remembers where to start the next read ahead by setting the `nextrio` inode field to the current location plus the size of the current cluster. The next two calls do nothing except return the page. Even the call for page 3 finds the data in memory because this data was prefetched. But we notice that this is the start of a new cluster and we start up the prefetch of 6, 7, and 8. The pattern repeats indefinitely, every third fault will start a prefetch three pages ahead.

Earlier, we said that, although the allocator tries to place a file contiguously on disk, it may not be able to do so because of fragmentation. This means that the cluster sizes sent back from `bmap` may vary at any point. In fact, an old file system will always send back a cluster of one block because of the rotational delays between each block. To insure that the read ahead code works regardless of cluster size, the code that sets up the next read bases its calculations on the returned rather than desired cluster size.

Write clustering implementation

The implementation of write clustering is contained in `ufs_putpage`. We handle writes by assuming sequential I/O and pretending that the I/O completed immediately (in other words, do nothing). If the sequentiality assumption is found to be wrong at the next call, we write the previous page out and then start over with the current page. If the assumption is correct, we keep stalling until a cluster is built up and then write out the whole cluster. The implementation relies on the page cache to hold dirty pages that `ufs_putpage` pretended to flush. The sequence of events is shown in figure 7.

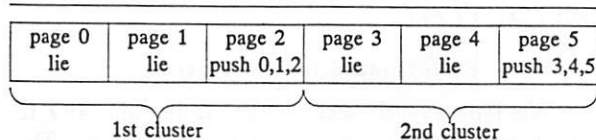


Figure 7 – Clustered writes with `maxcontig = 3`.

To implement write clustering, we added two more inode fields: `delayoff` and `delaylen`, as seen in figure 8. These new fields indicate the offset of the first page that was delayed and the number of pages delayed (in bytes), respectively.

We use these variables to detect sequential vs. random write patterns. If we do detect random writes, we write out the old pages between `delayoff` and `delayoff + delaylen` before restarting the algorithm with the current page; this is not shown in figure 8.

```

if (delaylen < maxcontig &&
    delayoff + delaylen == off) {
    delaylen += PAGE_SIZE
    return
}
find all pages from delayoff
to delayoff + delaylen
while (more pages) {
    bmap()
    start I/O for this cluster
    subtract that many pages
}

```

Figure 8 – Clustered write algorithm.

The fact that the allocator may not be able to allocate contiguously is reflected in the addition of the while loop. Note that this means we do not know if the file is allocated contiguously until we try to write out the cluster.

Unanticipated Problems

The implementation of clustering uncovered other problems in the system which are described here. Many of these can be traced to the interaction of the file and VM subsystems.

Page thrashing.

We thought that the file system was the only major bottleneck in I/O throughput, but in fixing it another problem area appeared: the paging part of the VM system. After reducing the file system overhead by clustering, we expected to be able to see throughput rates equivalent to the disk bandwidth. The throughput was lower than expected and we found that the VM system was the culprit. Pages were entering the system at a higher rate than they could be freed.

The unified VM system has only two ways of freeing pages: removing the backing store (unlinking the file) or running the pageout daemon. The pageout daemon implements (or tries to implement) a least recently used page replacement algorithm. The algorithm is the basic two handed clock and is explained in [Leffler]. The first hand of the clock clears reference bits and the second hand frees the page if the reference bit is still clear. The hands move, in unison, only when the amount of free memory drops below a low water mark.

Considering large sequential I/O, we can see that the pages just brought in are recently touched and as such will not be candidates for page replacement. This has the side effect of using all of memory as a buffer cache for I/O pages. For limited I/O, this is generally a good policy, but for large (greater than memory size) I/O this is a poor policy since it will replace all, potentially useful, pages with I/O pages that are unlikely to be reused. The VM system implements a least recently used (LRU) page replacement algorithm but for large I/O it

should implement most recently used (MRU).

Suppose we were to move an infinite amount of data through the system. If we have other users on the system, we don't want to disturb their pages or they won't be able to do any work. In this case, the best thing to do is to use and reuse a small number of pages, say the current cluster's worth. Unfortunately, this is not always the best thing to do or it would be the default in the system. If we used MRU for every file, we would effectively turn off caching, which is as bad as the original problem of destroying the cache.

We needed a compromise that would allow large I/O to go through the system with little impact but still leave in place the caching effects for smaller files. The compromise is inelegant and eventually the paging subsystem will be improved to address these issues properly. For now, we turn on *free behind* if the file is in sequential read mode, at a large enough offset, and free memory is close to the low water mark that turns on the pager.

Free behind is triggered in `rdwr` when the kernel unmaps the page. If the free behind conditions specified above are met, then the unmap will cause a call to `ufs_putpage` that will free the page. Free behind has the desired attribute that the process that is causing the problem is the process finding the solution. The pageout daemon no longer wakes up to free pages when the system is heavily I/O bound, since the I/O bound processes are doing it themselves. Having a process do the free behind in the I/O code path eliminates the overhead associated with switching to and running the pageout daemon.

Write limits or fairness

There is a fairness problem with `write` in the VM system. A single process can lock down all of memory by writing a large file (remember that `write` I/O is asynchronous; the kernel copies it and allows the user process to continue). In old UNIX systems, the buffer cache imposed a natural limit on the amount of memory that could be consumed for I/O. In the SunOS VM implementation, where all of memory is used as a cache, there is nothing to prevent a single process from dirtying every page. For example, a large process dumping core can cause the system to be temporarily unusable, since all the pages are essentially locked (they are dirty and in the disk queue which is the same as being locked down).

This is a basic fairness problem – the asynchronous nature of writes may be used to the advantage of one process, but it may be at the expense of other processes in the system.

Our solution to this problem is to limit the amount of data that can be in the write queue on a per file basis. We do this by adding what is

essentially a counting semaphore in the inode. Each process decrements the semaphore when writing and increments it when the write is complete. If the semaphore falls below zero, the writing process is put to sleep until one of the other writes completes.

The initial value of the semaphore has to be chosen carefully. If it is too large we return to the old problem; if it is too small, we will degrade both sequential and random performance. The sequential problem is exposed when we consider the I/O path as a pipeline. We need to feed the pipe at a fast enough rate that we never have any bubbles. For example, suppose we allowed only one write at a time in the queue. The first write would go down to the driver and the second would block, waiting for the first to complete. When the first completes, the second starts down, but this is too late. By the time the second request makes it out to the drive, there is a good chance that the drive will have rotated past the desired block.

The pipeline problem can be solved by allowing two or three outstanding writes, but this is still not good enough. There is another problem with random access. Consider a process that seeks to the beginning of the disk, writes a block, seeks to the end, writes a block, back to the beginning, writes a block, and so on until N blocks have been written. If we allow the disk queue to be infinitely large, then `disksort` will get a chance to sort the requests such that the system will seek to the beginning, write $N/2$ blocks, seek to the end, and write $N/2$ blocks. The effective I/O rate will be much higher in the case without a write limit than the case with a write limit of one. For this reason, we allow a fairly large (currently 240KB) amount of I/O per file in the disk queue.

The limit is currently set on a global basis for all processes. This is not as flexible as it could be. The write limit may be better implemented as a resource limit on a per process basis (see `getrlimit(2)`).

Performance Measurements

We ran several benchmarks, from pure I/O to multi-user time-sharing, to test out our work. The I/O benchmarks, as shown below, showed substantial improvements, but the time-sharing benchmarks improved only slightly.

We were a little disappointed with the time-sharing numbers until we examined the benchmark in detail. The benchmark, `MusBus`, was spending most of its time sleeping and the rest of the time running small programs such as `date(1)` and `ls(1)`. The largest I/O transfer done by `MusBus` was around 8KB which is the file system block size. In other words, `MusBus` didn't move any substantial amount of data.

	cluster size	rot delay	UFS version	free behind	write limit
A	120KB	0	SunOS 4.1.1	Yes	Yes
B	8KB	4	SunOS 4.1	Yes	Yes
C	8KB	4	SunOS 4.1	No	Yes
D	8KB	4	SunOS 4.1	No	No

Figure 9 – IObench run descriptions.

We use an internal program called IObench to show transfer rates. Figure 9 explains the configuration of each of four I/O benchmark runs. The hardware configuration is the same in each run, an 8MB, 20MHz Sparcstation 1, with one 400MB 3.5" IBM SCSI drive. We used a kernel that has variables that enable and disable the old and new code in an attempt to get an apples to apples comparison. The "A" configuration is almost identical to that shipped with SunOS 4.1.1; the difference is that the file system has been tuned to use 120KB clusters instead of 56KB clusters. The last configuration, "D," is a close approximation of a SunOS 4.1 installation; the file system has been tuned to make 1 block clusters with the standard 4ms rotational delay. The "B" and "C" configurations are similar to "D" but add some of the paging and fairness heuristics described in the section on unanticipated problems.

In the results shown below, the columns are headed by a three letter name indicating the type of I/O. The first letter means File system, the second letter indicates Sequential or Random, and the third letter indicates Read, Write, or Update. The difference between write and update is that in the update case the file's blocks have already been allocated.

	FSR	FSU	FSW	FRR	FRU
A	1610	1364	1359	383	452
B	805	799	790	369	431
C	749	783	784	366	428
D	749	722	718	370	545

Figure 10 – IObench transfer rates in KB/second.

Figure 10 shows the transfer rates, for the various I/O types, for four different software configurations. Since the numbers are hardware specific, we show and discuss the ratios below.

	FSR	FSU	FSW	FRR	FRU
A/B	2.00	1.71	1.72	1.04	1.05
A/C	2.15	1.74	1.73	1.05	1.06
A/D	2.15	1.89	1.89	1.04	0.83

Figure 11 – IObench transfer rate ratios.

In figure 11, we can see that almost all I/O rates improved, some slightly and some substantially. Predictably, the sequential I/O rates improved

about a factor of two. Reads are better than writes because the track buffer helps only reads. We made a tradeoff in favor of reads in not adding rotational delays between clusters. If the delays are present, the writes will improve slightly, but the reads will degrade slightly.

The random update (or write) numbers went down when compared to the generic 4.1 UFS. We made a tradeoff between performance and fairness in favor of fairness, which is explained in the section on unanticipated problems.

We used yet another internal benchmark for comparing CPU time. The benchmark is similar to IObench, in fact it shows identical I/O rates, but uses the mmap interface to avoid the copying of data from the kernel to the user. The IObench CPU times are dominated by the copy time and hence are approximately the same. Since we want to show the overhead of the new system versus the old, we used mmap. The cpu times in figure 12 show the seconds used by the CPU to read a 16MB file. The new UFS is approximately 25% more efficient in terms of CPU cycles. We believe that we can do even better; we explain how in the section on further work.

CPU	Notes
2.6s	4.1.1 UFS, no rot delays, 16MB mmap read
3.4s	4.1 UFS, rot delays, 16MB mmap read

Figure 12 – System CPU comparison.

Comparison to Related Work

Peacock's System V clustering [Peacock] is the most similar work we've found. The reasoning of reducing per byte overhead by doing larger requests is the same. Both designs try to improve performance by turning sequential I/O requests into larger sequential I/O requests. We believe that most of the following differences can be traced to starting with one base or the other, UFS versus the System V file system (S5FS).

- We depend on the FFS allocator to lay out the files contiguously. Originally we had planned to preallocate blocks, but we found that the allocator does such a good job that there was little to be gained by preallocation. The same is not true of the S5FS allocator. As Peacock pointed out, it is based on a free list that gets scrambled as the file system ages. Peacock was forced to rewrite the allocator to make use of the new bitmap free list. The rewrite caused on-disk format changes which were reflected in the file system utilities such as fsck, mkfs, etc.
- The UFS interfaces (ufs_getpage, ufs_putpage) are general enough that no changes were needed for clustering. Unfortunately, the same is not true of the S5FS interfaces (bread, bwrite). Peacock added

`mbread` and `mbwrite` to cluster the I/O while we were able to hide the clustering beneath the `ufs_getpage` and `ufs_putpage` interfaces.

- Our write algorithm is different, it starts a write each time a cluster boundary is crossed. Peacock's waits until the buffer cache fills up. The problem with waiting is that the system periodically flushes the cache to avoid file system inconsistencies in the event of a system crash or power failure. If the machine has a large buffer cache (large memory) then the flush may cause a proportionally large I/O burst. If the I/O were flushed to disk at each cluster boundary, the disks are kept uniformly busy, instead developing large disk queues. Smoothing out the disk queue will improve perceived performance since new requests will be serviced quickly.
- As described above, the SunOS VM system had no I/O heuristics. Peacock was able to use the buffer cache heuristics where we had to add them in order to prevent the pageout daemon from hogging the machine.

Further Work

Performance work is never finished; there is always one more refinement. In this section, we sketch out further work that could be applied to the file system. Some of these ideas have to do with clustering but others look at other ways of improving other aspects of file system performance.

Random clustering. Clustering is currently enabled only when sequential access is detected in the `ufs_getpage` routine. Certain access patterns, such as random reads of 20KB segments of a file, will not receive the full benefits of clustering. If the request is a read of a large amount of data, it is possible that the request size could be passed down to the `ufs_getpage` routine, which could use the request size as a hint to turn on clustering for what is apparently random access.

Bmap cache. The translation from logical location to physical location is done frequently and gets more expensive for large files because of indirect blocks. A small cache in the inode could reduce the cost of bmap substantially.

UFS_HOLE. Since UFS allows files to have holes, it is possible for `bmap` to return a hole. If we look back at the `ufs_getpage` algorithm (figure 2), we see that `bmap` is called even when the requested page is in memory. The reason for this call is that `ufs_getpage` needs to know if the requested page has backing store (i.e., is not a page of zeros from a hole in a UFS file). If the page has no backing store, then `ufs_getpage` must change the page protection bits to be read only. A read only page will fault when written, allowing UFS the chance to allocate the block to back the page. If the system did not enforce these rules, a write may appear to

succeed but later will find that there is no more space in the file system.

If UFS did not allow holes in files, we could bypass the `bmap` in all the cases that the page was in memory. One possible solution is to remember whether the file has holes and do the `bmap` only if the page is not in memory or if the file has holes.

Data in the inode. Many files are small, less than 2KB. Caching small files in the system causes fragmentation since the cache is made up of pages which are typically larger than the average file. We would like the caching effect without the fragmentation effect. This could be achieved by increasing the size of the inode in memory and caching small files in the extra space. This is already done for symbolic links if the link is small enough (the space normally used for block pointers is filled with the symlink data on the first access). Inodes are already cached in the system separately from pages which means that the system could satisfy many requests directly from the inode instead of the page cache. This would not work for `mmap()` since the data would not be page aligned.

Extents vs blocks. UFS maintains a physical block number for each logical block number. Given that UFS now allocates mostly contiguous files, there is a potential for substantial space savings by storing extent tuples of <logical, physical, length> instead of a long list of physical blocks. Unfortunately, this would mean an on-disk format change which is not acceptable for UFS. However, if this idea were coupled with the inode cache, large files could use the extra space as a `bmap` cache. To maximize the benefit of the space, the cache could be a cache of extent tuples.

B_ORDER. We would like to improve performance of UFS for the average user, not just the users who want high sequential I/O rates. One approach is to discard UFS in favor of a log based file system [Rosenblum]; this approach has merit. However, there are improvements that can be made to UFS today, and the installed base of UFS disks makes them worth considering.

A long standing problem with UFS is that it does many operations, such as directory updates, synchronously to maintain file system consistency on the disk. The file system uses synchronous writes to insure an absolute ordering when necessary. If there was a way to insure the order of critical writes, the file system would be able to do many operations asynchronously. The performance of commands like `rm *` would improve substantially.

We are considering adding a new flag, `B_ORDER`, that would be passed down to the various disk drivers. Requests in the disk queue with the `B_ORDER` flag may not be reordered by the driver, by `disksort`, or by the controller.

Summary

We have shown an enhancement that doubles the potential I/O rate of any UFS based file system. We described our implementation and the results of our implementation. The results show that the disk potential can be realized and also show that our method is less costly in CPU cycles than the old method.

Our approach was similar to that taken by extent based file systems, but differs in important ways: the extent size is variable, maintained by the file system, and is not exposed to the user. We believe that the user is rarely able to choose a correct extent size because there rarely exists a "correct" extent size. The optimal extent size varies based on many factors that may change during the life of an application. Even given that an extent based file system may be able to provide guaranteed throughput for the application that chose the optimal extent size, we believe that the enhanced UFS will provide better average throughput, since UFS is trying to allocate extents for all applications, not just the "smart" applications.

Acknowledgements

Many people contributed to this project. We would like to thank the following: Anil Shivalingiah, who explained the VM implementation over and over, Matt Jacob, for SCSI knowledge and the driver clustering implementation, Glenn Skinner, Bill Shannon, John Pope, Mark Smith, and David Rosenthal, for their helpful comments on this paper, Rich Clewett and Pat Townsend, for providing the hardware resources without which this project would have never completed, and the systems group environment at Sun Microsystems that made this work possible.

References

- [Bach] M. Bach, *The Design of The Unix Operating System*, Prentice-Hall, 1986.
- [Bobrow] D. Bobrow, J. Burchfiel, D. Murphy, and R. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM*, 15(3) March 1972.
- [Gingell] R. Gingell, J. Moran, and W. Shannon, "Virtual Memory Architecture in SunOS," *Proceedings of the Usenix Conference*, Summer 1987.
- [Kleiman] S. Kleiman, "Vnodes: An Architecture for Multiple File Systems in Sun UNIX," *Proceedings of the Usenix Conference*, Summer 1986.
- [Leffler] S. Leffler, M. McKusick, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.
- [McKusick] M. McKusick, W. Joy, S. Leffler, and

R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, 2(3) August 1984.

- [Moran] J. Moran, "SunOS Virtual Memory Implementation," *Proceedings of the European UNIX User's Group*, April 1988.
- [Organick] E. Organick, "The Multics System - An Examination of Its Structure" *M.I.T. Press*, 1972.
- [Rosenblum] M. Rosenblum and J. Ousterhout, "The LFS Storage Manager," *Proceedings of the Usenix Conference*, Summer 1990.
- [Patterson] D. Patterson, G. Gibson, and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Report No. UCB/CSD 87/391*, December 1987.
- [Peacock] K. Peacock, "The CounterPoint Fast File System" *Proceedings of the Usenix Conference*, Winter 1988.
- [Ritchie] D. Ritchie and K. Thompson, "The Unix Time-Sharing System," *Bell System Technical Journal*, 57(6), July-August 1978.
- [Thompson] K. Thompson, "Unix Implementation," *Bell System Technical Journal*, 57(6), July-August 1978.

Larry McVoy is currently a Member of Technical Staff in the Operating Systems Technology Department at Sun Microsystems. He received M.S. in 1987 and B.S. in 1985 in Computer Science from the University of Wisconsin at Madison. Since then, he has ported Unix to a super computer, brought up TCP/IP on machines ranging from 80386 to a super computer, added POSIX conformance to SunOS, and lectured at Stanford University on Operating Systems. Since joining Sun, he has been improving the performance of the VM and file subsystems of SunOS. He may be reached by electronic mail at lm@Eng.Sun.COM, by phone at (415) 336-7627, or by mail at MS 5-44, 2550 Garcia Ave., Mountain View, CA, 94043.

Steve Kleiman is currently a Distinguished Engineer in the Operating Systems Technology Department of Sun Microsystems. He received an M.S. in Electrical Engineering from Stanford University in 1978 and a B.S. in Electrical Engineering and Computer Science from M.I.T in 1977. He has been involved with the design and development UNIX and workstation architecture since 1977; first at Bell Telephone Laboratories and then at Sun. He was one of the developers of NFS, Vnodes, and the original port of SunOS to SPARC. His electronic mail address is srk@Eng.Sun.COM.

Smart Filesystems

Carl Staelin, Hector Garcia-Molina – Princeton University

ABSTRACT

Over the last few years tremendous strides have been made in CPU performance without corresponding strides in I/O performance. Consequently, future operating systems must be redesigned to minimize the impact of the I/O bottleneck. We present the concept of a *smart filesystem* as one that can dynamically and automatically tune itself to improve performance based on file access statistics it collects. We describe the iPcress File System, a prototype smart filesystem, and demonstrate a simple implementation of a disk data clustering technique. With this approach, active data is placed near the center of the disk, reducing seek times.

Introduction and Motivation

We motivate the need for new high performance file systems, and introduce the concept of a *smart filesystem* as a general technique for improving file system performance. We also present a prototype smart filesystem, the iPcress Filesystem which is being developed at Princeton¹, and evaluate the performance of a simple optimization: clustering active disk data near the center of the disk.

Over the last few years several trends in hardware development have emerged. These trends inviolate the basic price/performance tradeoffs that have been made in operating system design, and they imply that operating systems may have to be redesigned to conform with the current hardware environment. The two most important trends are the dramatic improvements made in CPU performance and the lack of any significant improvements in secondary storage (disk) performance.

Amdahl's Law is used to predict the impact of improving a single component on overall system performance. It can be stated as follows:

$$Speedup = \frac{s + c_0}{s + c_1}$$

where c_0 and c_1 are the times used by the slower and faster versions of the component and s is the time used by the rest of the system. If we evaluate the impact of CPU performance on system performance by letting s be I/O time and c be CPU time, we can see that the speedup is limited to (when

$c_1 = 0$):

$$Speedup = 1 + \frac{c_0}{s}$$

Amdahl's Law implies that I/O to secondary storage will soon bottleneck and that further improvements in CPU design will bring steadily diminishing returns.

Since most I/O is done through the file system in general purpose computing, we focus on improving the file system performance. Most common file systems (UNIX, MVS) were designed about 20 years ago under different price/performance constraints, so we expect to be able to improve file system performance. In particular, file systems (e.g., the UNIX file system) have been optimized for minimal space consumption and moderate performance. However, with file system performance becoming increasingly important to system performance, the file system should be optimized for maximal performance and moderate space consumption. Recently, there has been increased interest in file system development with several new file systems such as the Amoeba [13] and Sprite [8] file systems, which address several of the shortcomings of existing systems.

The design of the iPcress File System also addresses some of these shortcomings. Some of the techniques used are similar to those used by some of the new file systems, other are not. The most distinctive feature of iPcress is that it can automatically and dynamically modify its behavior and rearrange its storage based on file system use. For example, it can make caching decisions based on the likelihood that a file will be accessed in the future, or it can place frequently accessed files in the center of the disk. The key point is that the file system tracks (and records) both how individual files and the system as a whole are used, and it bases optimization decisions on that information. We refer to such an adaptive file system as a *smart file system*.

¹This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0456 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

iPpress File System

The iPpress file system has been implemented at Princeton University and is being used as a test-bed for file system optimization techniques. It is written in C++ and is currently running on a DECstation 3100. It is currently implemented as a single threaded user process running under ULTRIX 3.1, but in the near future we intend to migrate it to OSF/1 and transform it into a user-level multi-threaded server process.

iPpress was designed to be used as a high performance file system for a large general purpose computer with a large memory and many disks. The following features are currently implemented in iPpress:

- Statistical data on each file's access history.
- Large file-oriented disk cache.
- Variety of storage techniques and caching algorithms.
 - Cache whole file at file open for small and medium size files.
 - Simple block LRU caching for large randomly accessed files.
- Variable size blocks in both memory and secondary storage.
- Multiple disks (devices) within a single file system.
- Single threaded user-level NFS server.

Features not yet implemented:

- multi-threaded user-level file system.
- Predictive read-ahead/flush-behind for large sequential files.
- Database technology for reliability and (fast) recovery.

To achieve high disk transfer rates, it is desirable to allocate files contiguous disk space (these contiguous blocks are often called "extents") and to store their images also contiguously in memory. However, extent-based file systems have trouble managing growing files, and at least one such file system (IBM's MVS file system) requires that space be preallocated for each file. When a file grows beyond the size of the extent, the file system must either split the file over several extents or allocate a new, larger extent for the whole file. In the first case, future performance is potentially diminished, and in the latter case the overhead of moving the file is costly. Also, user tendency in an environment requiring pre-allocation is towards (dramatic) over-estimation of file size, leaving empty space at the end of each file.

We avoid these problems by using a variable size blocking scheme, as in the [7]. The system manages blocks of sizes 512 bytes, 1k, 2k, 4k..., 32k. Free blocks are managed using the buddy

system; for example, adjoining free blocks are coalesced into a single one of the next larger size. Except for very large files, files fit in a very small number of blocks, with little wasted space. An entire file can be transferred in or out of memory with a very small number of I/O operations.

The availability of large amounts of memory in current computers makes it desirable to perform caching at the file level as opposed to the block level. For this, caching decisions are split into two categories, *staging* which involves moving data from disk to the cache, and *flushing* in the reverse direction. Cache staging decisions are done on a per file basis, while flushing is done on a global basis using a LRU block algorithm. This way the system may use per-file information for predictively caching files.

The single most important object in iPpress is the file object. The file object provides the basic file operations. There are many different types of file objects, each of which provide the same basic operations, such as "read" or "write." In addition, each file object type can be described by two or more "properties." The two basic "properties" are storage method and caching method. There are several types of these methods, and more can be added easily. File objects are allowed to dynamically change type as the situation demands. In addition, each file object keeps track of its own access history information.

The storage file property determines how a file is stored on disk. For example, small files (less than 32k bytes) may keep their data in the inode, while big scientific data files may be kept in multiple blocks on several devices. It will be also possible to have a file reliably stored, by using redundant data spread over several devices. The caching property determines how the file is cached and it stages buffers from secondary storage into the cache. Again, there is a large variety of techniques available, each of which is appropriate for a different class of file. Some files might be completely staged upon file open, while others, such as large database files, may use more sophisticated "run detection" techniques.

One interesting feature of iPpress is the fact that the file system data structures, such as the block free list, are kept in files. In iPpress everything is a file, including the indexed table of file headers (inode table). This is an extension of the UNIX philosophy that "all files are simply a stream of bytes." For example, in UNIX, the inode table is contained in a fixed location on disk, and its size is defined at the time the file system is created. In iPpress, the UNIX inode table is a file which contains an array of inode records, and it may grow and shrink just like any other iPpress file. It may also use the options available to other iPpress files, such as reliability. For more information regarding the design of iPpress see [11].

Optimizations

Conventional file systems do not recognize the fact that files are used with wildly varying probabilities, and that these probabilities are reasonably predictable. Consequently, all files are treated alike, and nothing is done to improve performance for those few active files which dominate file system performance. As we have stated, iPpress uses statistical historical information to optimize performance by either reorganizing data placement on disk, or by modifying file caching strategies.

We were motivated to utilize statistical information by a detailed study of file access patterns we performed at Amdahl Corporation [12]. Trace data was produced at two real, large customer locations using IBM's MVS/XA operating system. The traces were generated using IBM's System Management Facility (SMF), and included data of file opens and closes, the number of I/O's per session, the number of tracks allocated to the file, and so on.

Analysis of these traces showed, as expected, that file access patterns are highly skewed. In a typical three day run, 85% of the I/O's were directed to 10% of the files; the remaining 15% of the I/O's was to the other 90% of the files. The 10-90% breakdown is of files that were accessed during the three day run. In addition there is a very large number of files that were never accessed during the run. Since these files were not recorded in the traces, it is difficult to know exactly how many files were not accessed. However, we estimate that the accessed files were at most 20% of the total. Thus, the files that received 85% of the I/O's constituted at most 2% of the file system.

Skewed data access patterns have been observed in other studies. [12, 14, 1, 6] have looked at file access patterns in the IBM MVS system, while [9,5,4] have looked at the UNIX system. [4] shows that roughly seventy percent of read-only and eighty percent of write-only accesses are whole file transfers; most opens are to files which are opened hundreds or thousands of times, and most files were opened less than 10 times a week. [14, 1, 6] have demonstrated that file accesses are highly skewed, and that most file system activity is concentrated on a relatively small fraction of files. [9] shows that in UNIX most files accesses are whole file transfers and the read/write ratio is between 80/20 and 65/35.

Unlike other studies, we also looked at how the likelihood of access varied over time. For this we introduced the notion of *file temperature*. In the cache related literature, the terms "hot" and "cold" are used to denote objects which are accessed heavily and lightly respectively. We extend this notion to files and define the temperature of a file as the number of I/O's performed on the file for a particular time period divided by the file size.

Our results indicate that file temperatures drift slowly. That is, the hottest files tend to stay hot over a time frame that can be measured in days. In one experiment, for example, we measured file temperatures for a one day period. In this case, 1% of the files received 85% of the I/O's. (Again, this is 1% of the files accessed that day, not of the total number of stored files.) The same system was traced for a second day, tracking the file accesses to the files that had been the hottest 1% the previous day. It was observed that 70% of the I/O's were still directed to those files. Fifteen percent of the I/O's had "drifted" to other files not in this group. After three days, 50% of the I/O's was still directed to the same set of files.

This shows that it may be advantageous for a file system to periodically reorganize its data by file temperature. One idea for such optimization is to cluster active disk data together, and if possible to place it in the center of the disk. This technique is well known, and manual placement of files in this fashion has been documented as a standard optimization method for large (MVS) installations [2]. Also, in the MacIntosh environment there is a new product called DiskExpress II which monitors file activity and rearranges data automatically each night so the active files are clustered at the start of the disk. (This is not part of the file system; it is a stand alone utility.) Clustering active files together has several advantages. It can reduce seek times dramatically: If it is likely that consecutive accesses are to hot files, then it is likely that they will be to blocks in the center of the disk, reducing the expected seek time. In a heavily loaded system with long disk request queues, the active cylinders in the center will tend to have multiple requests. Cylinders on the periphery will have none. This means that in a single disk rotation, several requests can be satisfied, increasing the throughput.

Another idea (not yet implemented in iPpress) is to do disk load balancing. Research has shown [3] that in most systems some disks receive much higher I/O loads than other disks. Even systems that are balanced with respect to an entire day's activity, tend to be strongly unbalanced during smaller periods. To avoid this, hot files can be distributed across available drives, so that all devices are utilized, and average disk delay is reduced. Again, this is not a new idea. For example, POPL [15] is a load balancer developed for the IBM MVS environment which analyzes traces of file system activity to determine file temperature. Unfortunately, processing the (large) system logs is time consuming. Also, since POPL is not part of the file system it cannot dynamically balance disk load via techniques such as placement of temporary files.

Our goal is to incorporate optimizations such as clustering and balancing into the file system itself, so that statistics gathering and the optimizations can

be done automatically and on a routine basis. There are several key issues that must be addressed for this: how to reorganize the data to improve performance; how to keep overhead to a minimum so that performance gains are not lost; and how to move the least amount of data to get the best performance gain. In the rest of this paper we discuss an implementation of the disk clustering optimization, and report on experiments that illustrate the potential gains.

Disk Data Clustering Implementation in iPpress

A simple "proof of concept" disk data clustering algorithm has been added to the iPpress file system. It is a simple batch-oriented prototype and not a complete production version. The clustering algorithm uses file activity statistics kept by iPpress on each file to rank files by their relative activity. Currently iPpress uses the number of bytes transferred to or from the disk on behalf of the file over the number of bytes allocated to the file as the measure of file activity. The system moves all files with no accesses (dormant files) as close to the edges of the disk as possible, and then it moves files as close as possible to the center of the disk based on the ranking (according to file temperature).

The disk space is divided into *regions* or *buckets* of unequal size. The regions are symmetrical about the center of the disk, and they grow exponentially in size as one moves away from the center. For example, the center region consists of a single cylinder, and the next region consists of two cylinders, one on each side of the center region. Blocks within a given region are considered to be "equal" with respect to their distance from the center of the disk. Consequently, the file system tries to place a file within a region according to its temperature.

Ideally, one would like to place the files in strict temperature order on the disk, i.e., the hottest file in the very center, then the next hottest and so on. This would make clustering payoff the most, but would be expensive to achieve. The bucket approach gives more flexibility for placement, but how much will we pay in decreased performance? To answer this question before implementing clustering, we performed a detailed analysis [Staelin90Clustering]. We modeled the DEC RZ55 disk that was used by iPpress as closely as possible, using a non-linear seek cost function. We then simulated various implementation options, and drove the simulation using access distributions based on our measured patterns.

The simulation results showed that a relatively small number of buckets yielded performance close to that of a perfect placement policy. For six buckets, performance was already within a few percent of optimal. Thus, in iPpress, we implemented a system using eight buckets. However, the number of

buckets is easily changed and may be increased if necessary.

Incidentally, our simulation showed that performance gains would be significant even if file temperatures drifted over time. For example, say disk utilization when files are perfectly placed and there is no drift is U_p . When the files are placed at random on this disk utilization is U_r . Thus, the best case gains for clustering are then $U_p - U_r$. Next we modeled a system where 10% of the files had drifted, i.e., 10% of the files were randomly placed. In this case the utilization gains were still 80% percent of $U_p - U_r$. With 20% drift, gains were still 60% of optimal. This means that even if reorganizations are performed once a day, gains due to clustering can be significant.

Returning to our implementation, the disk reorganization algorithm is a four stage algorithm. The first stage scans the file system, moving all files with zero temperature to the edge of the disk, and building a table of file temperatures and addresses for the remaining files. At the end of the first stage, this table is sorted by descending temperature. During the second stage, the system determines the optimal placement (bucket) for each file. The third stage is used to move files that are too close to the center out towards the edge of the disk. The system scans over the file system in order of increasing temperature placing each file no closer to the center than its optimal bucket. At the end of this scan, cold files that are too close to the center will be moved out towards the edge, but hot files that are near the edge may not yet have room to move towards the center. The fourth stage will make sure that each file is in its optimal place. It scans over the files in order of decreasing temperature. However, when a file cannot be successfully placed in its optimal location, the system moves files with smaller temperature belonging to the same bucket down a bucket until the current file can be successfully placed in the proper bucket. If the current file is the last file in a bucket, the system only makes a single attempt to place the file in the correct location.

Benchmark

We evaluated the data clustering facility with a new benchmark FSBench, which emulates the strongly skewed nature of file access patterns. The benchmark consists of two programs: the *generator* which creates a table of file access probabilities for a file system, and the *driver* which uses this table to read data from the file system with skewed probability.

The generator takes as input a file system, a parameter that describes what fraction of the system will be dormant, the target number of mega bytes of data to be read during an experimental run, and a distribution that describes the access pattern to the

active data. As discussed in Section 3, a significant fraction of file systems tends to be dormant, i.e., not accessed at all. For instance, from discussions with L. Davis, one of the designers of the DiskExpress clustering system for McIntoshes, it seems that their systems often have 80% of the files dormant. The generator models this by not making accesses to the dormant fraction of the file system. For the files that are accessed (selected at random), the generator produces a table of file names and a number of times each should be accessed. The frequency of access is generated following the distribution given as input to the generator. The input distribution, which comes originally from experimental observations, specifies how hot each block is. However, it is guaranteed that each file that is not dormant will be accessed at least once by the driver during the experimental run.

In our experiments we used an experimentally derived distribution as input for the generator. This distribution was obtained in [14] by tracing system activity on a large MVS system over a one week period. (Similar to our study whose results are reported in Section 3.) Only those permanent files accessed during the tracing period appear in the distribution. The distribution is strongly skewed, with almost 20% of the total I/O during the week going to 0.5% of the disk space. In addition, nearly 60% of the I/O is directed towards 5% of the space, 88% of the I/O goes to the hottest 20% of the disk space, and 98% of the I/O goes to 50% of the space, leaving 2% of the I/O for the remaining 50% of the disk space.

There is one limitation to this benchmark: unless the number of bytes accessed during each experimental run is several times larger than the entire file system, the resulting distribution created by the generator will not be as skewed as the input distribution. This is due to the fact that each non-dormant file must be accessed at least once during each experimental run.

The second program in our benchmark, the driver, reads the table of file names and number of accesses and uses it to govern the sequence of file accesses. Each time it accesses a file it reads the whole file. During execution, it randomly chooses a file from the table to be accessed, reads the file, and decrements the number of accesses left for the file. The probability of choosing a particular file is the number of accesses remaining for that file over the total number of remaining accesses. Thus, at the end of a run, all files have been accessed the specified number of times. The driver reports the time necessary to process all of the given accesses.

Performance

We evaluated the performance of data clustering using FSBench. In order to measure disk performance improvements, we added monitors to iPpress which measure the cumulative time spent doing disk

I/O and the number of disk operations.

Each run of the experiment measures the performance improvement after a single reorganization. First, the system copies a file system tree into a new (empty) file system, then the generator creates the table of file accesses to be used by the driver for the rest of the run. At this point the driver accesses the file system the first time. Once the driver is finished both the driver and the file system report the first set of performance results, then there is a pause while the file system reorganizes itself. During the reorganization, iPpress utilizes statistics gathered during the first driver run to identify the temperature of each file. The most frequently accessed files are placed in the center of the disk, and so on with the reorganization. Finally, the driver accesses the reorganized file system again and the final performance results are collected from the driver and iPpress.

We used a variety of different file systems of differing sizes: 50MB, 70MB, and 210MB. The file systems consisted of subsets of a system tree containing system and kernel sources, objects, and documentation. The full 210MB tree had nearly 16,000 files. We configured FSBench to transfer 150MB per experimental run. However, for the 210MB case FSBench was transferring between 330MB (for 0% dormant) and 160MB (for 90% dormant) due to FSBench's restriction regarding the fact that all active files must be accessed at least once during the run.

The experiment used two computers, the iPpress NFS server, and the NFS client. Both machines are DECsystem 3100's with local disk and they are on the same thick-wire ethernet subnet. The subnet is one of Princeton's primary departmental subnets, supporting over sixty X-terminals and machines. During some of the experiments we experienced packet collision rates of over 10%, so we ignored the driver's performance results.

The disk used in the experiments is a DEC RZ55 330MB disk. Physically, it has 512 byte sectors, 36 sectors per track, 15 tracks per cylinder, and 1224 cylinders, and it is connected to the machine via a SCSI bus. Its platter rotates at 3600 rpm, so the average rotational delay is 8.3ms. The average seek time is 16ms, but unfortunately more detailed information regarding seek times is not available. iPpress accesses the disk via the UNIX raw device, which has roughly 310MB available. The mapping of the UNIX blocks to physical blocks is not visible to iPpress.

Table 1 presents the speedup in time per I/O operation (disk read or write) as a function of the scenario (dormant fraction, file system size). The speedup is computed as $(T_0 - T_r)/T_0$ where T_0 is the time per I/O operation spent by iPpress during the first driver run, and T_r is the time per I/O operation during the second run, after reorganization.

Note that T_r and T_0 include all I/O time per operation, including CPU time used by the disk driver, rotational delay, seek time, and transfer time. Data clustering in our experiments only improves the seek time. For each data point in Table 1, there may be about a 10% variability. For example, for a 50MB system with 80% dormant, Table 1 reports 16.3% speedup, which is the average of four runs. The standard deviation on these four runs was 0.6%. However, due to time constraints most of the other data points are the result of either a single run, or at most two runs.

The trends in Table 1 can be summarized as follows. As the fraction of dormant data increases the performance benefits of data clustering improve, since the disk head moves over shorter distances. As the size of the file system increases, the disk fills up and the amount of active data increases, so the performance improvements degrade slightly.

The speedups are due to decreased seek times alone. As we pointed out, the physical I/O time for each access consists of the seek time, rotational delay, and transfer time. In our case the average seek time is 16ms, the average rotational delay is 8.3ms, and the average transfer times vary between .5ms (for a 512B block) and 29.5ms (for a 32kB block). For an "average" block of 8kB, the transfer time is 7.4ms., so the seek time only accounts for 50% of the total physical time. Thus, we can interpret the average speedup in disk access time shown in Table 1 as being roughly half of the average improvement in seek times. That is, if Table 1 shows a 10% improvement, the seek times should have been reduced by at least 20%. Our benchmark issues requests one at a time and hence does not model a heavily loaded system. If the system were heavily loaded with long disk queues, one would expect speedups beyond what Table 1 shows, as clustering makes it possible to satisfy more and more requests in a single disk rotation.

As there is more data in the file system, the measured speedups due to reorganization drop slightly. We believe this is due to both the drop in the probability that an access is to the same cylinder as the last request and changes in the experimental file access distribution. When there is relatively little active data, the number of active cylinders is

small (roughly 40 active cylinders for the 50MB file system 80% dormant case), so the probability of requests going to the center cylinders is non-negligible (especially since the center cylinders are far more active even than the outer cylinders of the 40 active cylinders). However, the larger cases have so many active cylinders that this probability drops to near zero. For example, the 210MB file system with 80% dormant data has nearly 170 active cylinders.

As the amount of active data increases and the target number of mega bytes transferred by the benchmark is constant, the ratio of data transferred from cold files (accessed just once during each run) to data transferred from hot files diminishes. Consequently, the fraction of time spent transferring hot files decreases, so the relative benefit of reorganizing the data is reduced. However, this is a limitation of the current configuration of the benchmark, and not of the reorganization facility. The primary reason the benchmarks were not rerun with larger configuration parameters is that such experimental runs require several hours to a day each.

As the fraction of dormant data increases, the performance benefits of reorganizing data improve, almost doubling by the time 90% of the data is dormant. We believe this is attributable to the fact that large portions of the disk (both edges) are essentially never visited. However, most performance benefits are not evident until over half of the data is dormant. This is probably due to the non-linear nature of seek times (for modern constant-acceleration seek arms). In the event that many installations have roughly 80% of their data dormant, this is not a problem.

The time needed to reorganize the data is an important figure for user installations. For the full reorganization of the 210MB file system iPCress needed up to two hours to complete the reorganization. However, this is starting with a disk that has the data distributed completely randomly over the surface of the disk. In normal operation, most of the data would already be in the correct location from the previous day's reorganization. Consequently, reorganizations should take much less time than we report.

File System Size	Dormant Fraction			
	0%	50%	80%	90%
50 MB	10.5%	13.1%	16.3%	16.3%
70 MB	11.1%	12.9%	16.0%	17.0%
210 MB	6.8%	7.6%	9.5%	10.7%

Table 1: I/O Speedup

Conclusions and Future Work

We have presented the concept of a *smart filesystem* as a file system which optimizes its performance based on file usage statistics. We have described several possible optimization techniques which may be used by such file systems, and we have analyzed the implementation and performance of clustering active disk data in the center of the disk. We have shown how this alone may improve disk performance by six to eighteen percent.

In the future we hope to implement some of the other optimizations, such as disk load balancing. However, more research analyzing basic file access patterns should be done so that more optimizations may be proven feasible.

Acknowledgements

We would like to thank Dick Wilmot and Amdahl Corp. for the cumulative file system statistics used by FSBench.

References

- [1] K. Baclawski, S. H. Contractor, and R. Wilmot, "File Access Patterns: Self-Similarity and Time Evolution." College of Computer Science, Northeastern University, 1989.
- [2] M. G. Baker, "DASD Tuning -- Understanding the Basics," in *Proceedings CMG*, CMG, December 1989.
- [3] A. L. Bastian, J. S. Hyde, and W. E. Langstroth, "Characteristics of DASD Use," in *Proceedings CMG*, CMG, December 1981.
- [4] R. Floyd, "Short-term File Reference Patterns in a UNIX Environment," Tech. Rep. TR-179, Computer Science Department, University of Rochester, Rochester, NY 14627, March 1986.
- [5] R. Floyd, "Directory File Reference Patterns in a UNIX Environment," Tech. Rep. TR-177, Computer Science Department, University of Rochester, Rochester, NY 14627, August 1986.
- [6] M. Henley and I. Y. Liu, "Static vs Dynamic Management of Consistently Very Active Data Sets," in *Proceedings CMG*, CMG, December 1987.
- [7] P. D. L. Koch, "Disk File Allocation Based on the Buddy System," *ACM Transactions on Computer Systems*, vol. 5, no. 4, ACM, November 1987.
- [8] M. N. Nelson, B. B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, ACM, February 1988.
- [9] J. K. Ousterhout, et al., "A Trace-Driven Analysis of the UNIX 4.2BSD File System," in *Proceedings for the 10th ACM Symposium on Operating System Principles*, ACM, 1985.
- [10] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," in *Proceedings Winter 1990 USENIX*, 1990.
- [11] C. Staelin and H. Garcia-Molina, "File System Design Using Large Memories," in *Proceedings Fifth Jerusalem Conference on Information Technology*, October 1990.
- [12] C. Staelin, "File Access Patterns," Tech. Rep. CS-TR-179-88, Department of Computer Science, Princeton University, Princeton, NJ 08544, September 1988.
- [13] R. van Renesse, A. S. Tanenbaum, and A. Wilshut, "The Design of a High-Performance File Server," in *Proceedings of the 9th International Conference on Distributed Computing Systems*, June 1989.
- [14] R. Wilmot, "File Usage Patterns from SMF Data: Highly Skewed Usage," in *Proceedings CMG*, CMG, December 1989.
- [15] J. Wolf, "The Placement Optimization Program: A Practical Solution to the DASD File Assignment Problem," in *Proceedings SIGmetrics*, ACM, 1989.

Carl Staelin is a graduate student in the Department of Computer Science at Princeton University. His interests include operating systems, computer architecture, and distributed systems. He is currently finishing his doctoral thesis on high performance file systems. He received a B.S. degree in Electrical Engineering and a B.A. in Mathematics from the University of Rochester in 1985. He also received an M.A. in Computer Science in 1987, and expects to finish his Ph.D. in Computer Science at Princeton in 1991. His email address is staelin@cs.princeton.edu, and his U.S. Mail address is: Department of Computer Science, 35 Olden St., Princeton NJ 08544-2087. Staelin is a member of USENIX, the ACM and IEEE.



Hector Garcia-Molina is Professor in the Department of Computer Science at Princeton University, Princeton, New Jersey. His research interests include distributed computing systems and database systems. He received a BS in electrical engineering from the Instituto Tecnológico de Monterrey, Mexico, in 1974. From Stanford University, Stanford, California, he received in 1975 a MS in electrical engineering and a PhD in computer science in 1979. Garcia-Molina is a member of the ACM and IEEE.



Lessons Learned Tuning the 4.3BSD Reno Implementation of the NFS Protocol

Rick Macklem – University of Guelph

ABSTRACT

Since its introduction by Sun Microsystems in 1986, the NFS protocol has become the defacto standard distributed file system protocol for Unix based workstations. Most of these Unix implementations are based on the reference port provided by Sun Microsystems. Research published to date on NFS performance has focused on the problems of NFS server write performance and NFS server performance characterization. This paper discusses other performance and implementation aspects of NFS observed while tuning a rather different implementation of the Sun NFS protocol for Unix. Aspects of performance related to differences in caching mechanisms, the use of different RPC transport protocols and techniques that minimize memory to memory copying are explored. In particular, the notion that TCP transport would provide unacceptable performance for NFS RPCs is shown to be unfounded.

Introduction

There are several aspects of the 4.3BSD Reno implementation of Network File System (NFS) that set it apart from the Sun reference port.¹ In the 4.3BSD Reno implementation, particular emphasis has been put on caching mechanisms, network transport layer independence and the avoidance of memory to memory copy operations. To minimize memory to memory copying and retain network transport layer independence, the NFS remote procedure call (RPC) requests and replies are handled directly in mbuf² data areas. Network transport independence permits experimentation with running NFS over other protocols, including TCP. As such, it was felt that by benchmarking this implementation of NFS, we could gain insight into various aspects of performance that have not yet been adequately addressed.

This paper describes the results of benchmarking and tuning in three major areas:

- Server CPU overheads
- Effects of transport protocols
- Effects of different caching mechanisms

In Section 1, a brief overview of the NFS protocol is presented, followed in Section 2 by an overview of the 4.3BSD Reno NFS implementation.

¹Along with the published NFS specification, Sun Microsystems licensed a reference port of NFS which forms the basis of most commercially available NFS systems. Since I have no access to this code, information about its structure was gleaned from a variety of publications. Apologies for any inaccuracies w.r.t. this port.

²mbuf is the Berkeley Unix structure for handling network buffers.

Section 3 discusses techniques used to reduce server CPU overhead. Section 4 compares the performance of NFS over a variety of transport protocols operating on three different internetwork topologies. Following this, a comparison in Section 5 of 4.3BSD Reno NFS with Ultrix NFS is used to identify significant differences related to caching mechanisms. The conclusion summarizes the results and suggests areas of distributed file system performance that require further investigation.

1. Overview of NFS Protocol

The NFS protocol is a remote procedure call (RPC) based distributed file system that does I/O at the level of logical blocks of files. These data blocks start at an arbitrary byte offset and range in size from 1 to 8192 bytes. The server is stateless, which implies that RPC requests are atomic operations where all request related information must be stored in the RPC request. The stateless server concept was used so that crash recovery is trivial. However, there are some obscure implications on performance in the areas of client cache consistency and write policy.³ The write policy for NFS is asynchronous for full blocks and delayed when partial blocks are written. The delayed writes must be pushed when the file is closed and are also pushed every 30sec for most Unix implementations. Cached data consistency is maintained with the server by checking that the file's modify time has not changed since the cached data was read from the server.

³Write policy defines the client action when a write to a remote file is done. It may be write through which implies: do the write RPC and wait for the reply before returning from the system call. Asynchronous, implies start the write RPC but do not wait for its completion. Delayed means do the write RPC sometime later.

Since most implementations also cache file attributes for a few seconds, this implies that cached data will be consistent with that of the server to within a few seconds. However, the stateless server does not know about any delayed writes to a file from other clients. By pushing delayed writes on close, NFS maintains a close/open consistency criteria when more than one client read/write shares a file. That is, a client opening file "X" for reading after another client that was writing to file "X" does a close, is guaranteed to see those changes.

The NFS RPCs are done using Sun RPC, which stores all fields of the requests and replies in an architecture-independent data format, called the external data representation (XDR). For the Sun reference port, a user mode runtime library that implements these layers, was ported into the kernel, and NFS was implemented using this library interface.

2. 4.3BSD Reno NFS Implementation

The 4.3BSD Reno NFS is implemented in the kernel without the use of any XDR or RPC interface layers. All NFS RPC requests and replies are constructed and decomposed directly in mbuf data areas using two macros `nfsm_build` and `nfsm_disect`. These two macros are then used by higher level functions and macros to access the fields of the NFS RPC request and reply packets. Most of the translation to/from XDR is done by inline code, except for a few special cases that are handled by functions. There were two reasons for this approach, namely to:

- Avoid the use of a buffer that would have to be copied into an mbuf list.
- Avoid the need for a special type of mbuf that might not work well with transport protocols other than UDP.

Once the request or reply has been converted into an mbuf list, the list is passed onto the socket interface code which deals with the vagaries of the various types of sockets. For datagram sockets, the client side provides round trip timeout (RTO) estimation and requests retransmission upon timeout. For stream sockets such as TCP, it maintains the connection and provides record marks between each RPC request/reply, along with concurrency control on the socket I/O routines.

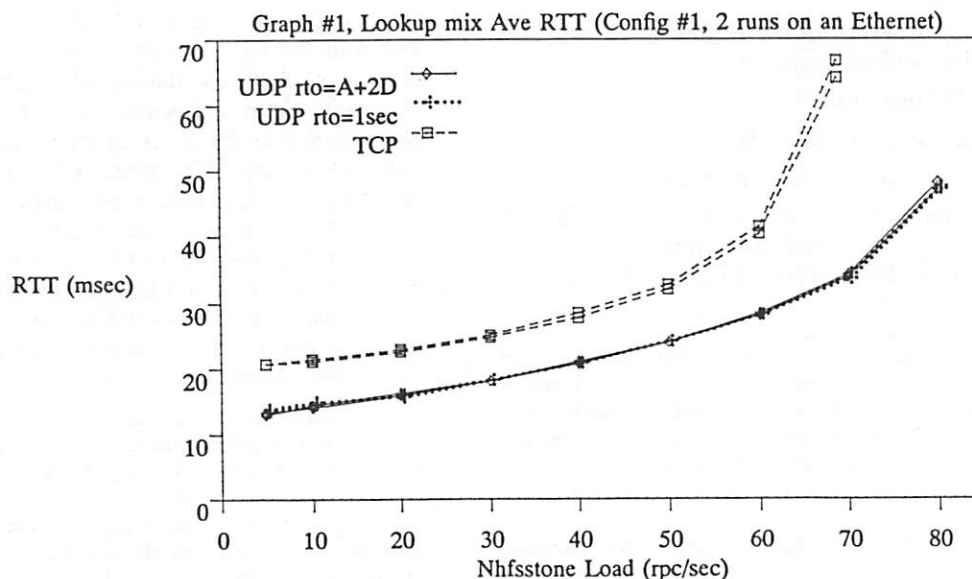
Caching is done for name lookups, data blocks and directory blocks, using the VFS caching mechanisms which are discussed in greater detail in Section 4. The client side cache consistency is controlled by the file/directory modify time, and cached data is flushed whenever the modify time changes, as reported by the server. The file attributes are cached and time out five seconds after being updated from the server. This appears to be similar to the level of consistency that was observed experimentally on a SunOS NFS client.

3. Server Structural Changes and CPU Overhead

Most current NFS servers tend to be CPU bound, which makes minimizing server CPU overhead of interest. To study this, the kernel of a system that was running under heavy NFS server load was profiled to identify bottlenecks. It was observed that over a third of the CPU cycles were being used by the low level network interface handling code. In particular, the routine that copied the mbuf data areas to the network interface's transmit buffers was at the top of the CPU utilization list.

In an effort to reduce CPU overhead in the network interface code, two changes were made:

- Network interface buffer handling was modified to allow the mapping of a packet to two noncontiguous buffers for transmission, one for the IP



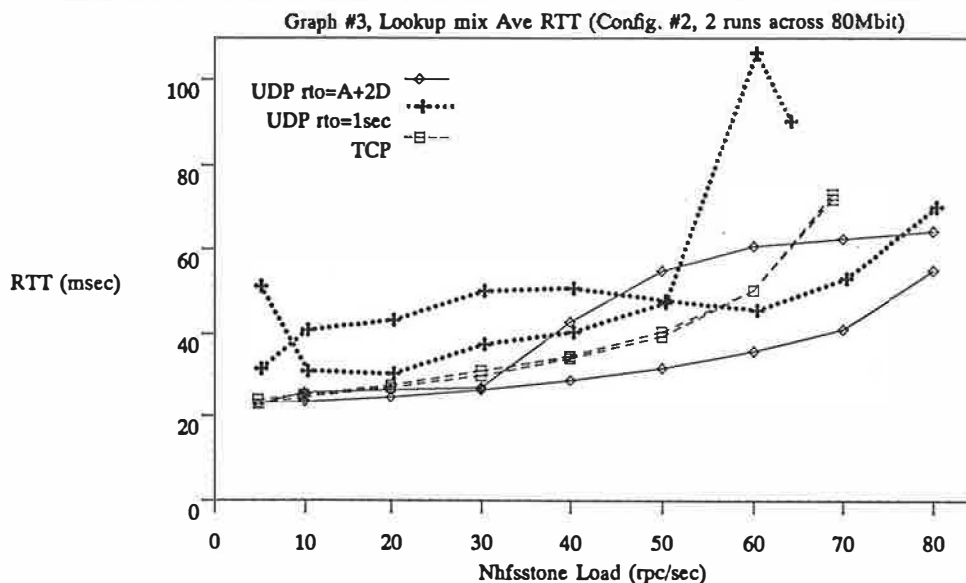
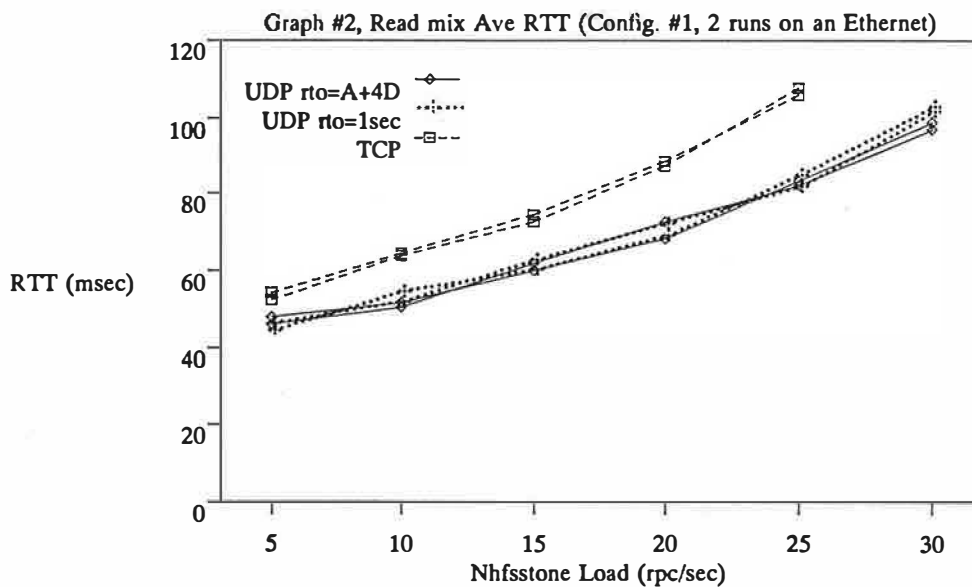
fragment header and the other for the mapped mbuf data clusters. This allowed the *copying* of mbuf clusters to network interface buffers by page table entry swaps instead of by actual memory to memory copying.

- The network interface device driver was modified to remove the transmit interrupt service routine. Since this routine simply released buffers and updated I/O statistics, it was possible to disable transmit interrupts and perform the operations in the transmit startup routine, reducing the number of network interface interrupts. [Jacobson89] The transmit startup routine was also fine tuned by careful use of register variables and unrolling of loops.

After the above changes, CPU overhead was reduced by approximately 12%. Most of this was a reduction in memory to memory copying. Since

memory to memory copy bandwidth has not grown with MIPS rate for many recent computer systems [Ousterhout90], this may be even more significant on newer hardware architectures.

At this point, the CPU bottlenecks were the network interface startup routine, the internet checksum calculation routine and the routine that copies data between the buffer cache and mbuf clusters. Since the first two bottlenecks have already been fine tuned, the only area that deserves further attention is the third. It may be possible to avoid the buffer cache to mbuf cluster copying by implementing a mechanism where page clusters in the buffer cache may be borrowed as mbuf page clusters and returned after network transmission. This was not done, due to the complexity of the code, but is a possible area for further work.

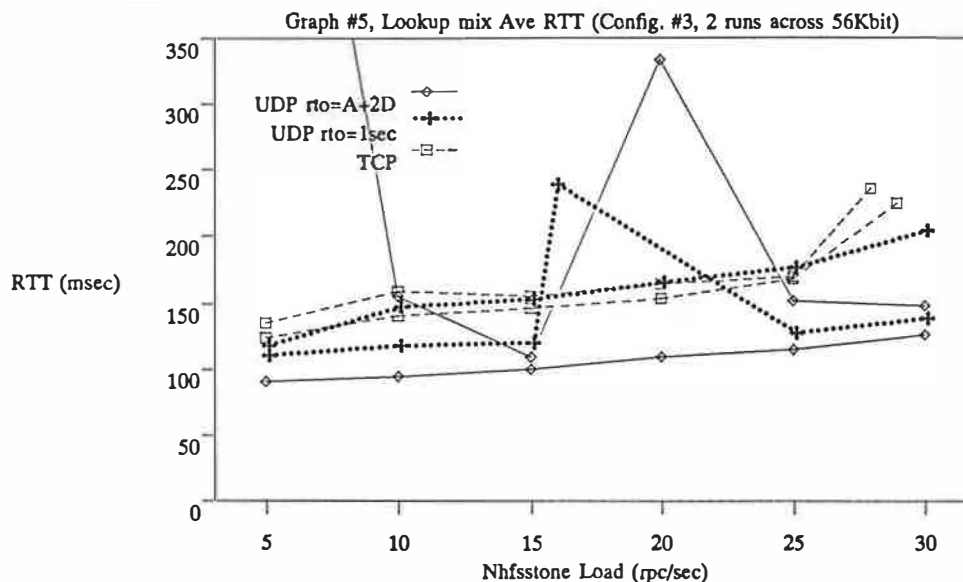
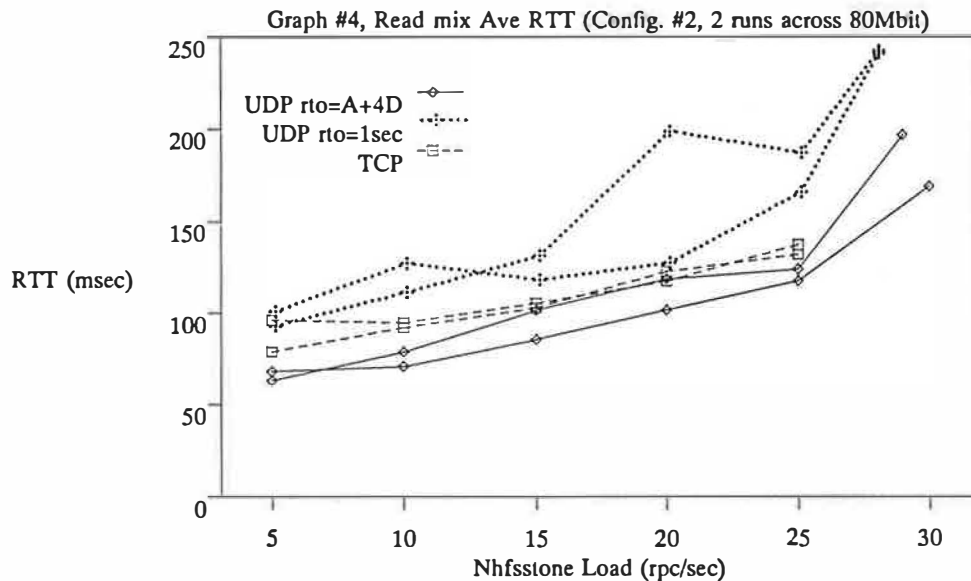


4. RPC Transport Issues

The NFS protocol normally runs on top of UDP transport where each RPC request and reply is packaged in one UDP datagram. Since UDP datagrams are not delivered reliably, the NFS client sets a retransmit timeout (RTO) when a request is sent and resends the request if the RPC reply is not received within the RTO. The initial RTO is set to a constant value defined at mount time and is backed off exponentially upon retransmits. This transport mechanism is adequate when the client and server reside on the same high bandwidth LAN cable, but performance has been observed to deteriorate over more complex internetwork connections. [Nowicki89] This problem stems in part from the fact that an 8Kbyte read or write RPC must be transmitted as IP fragments the size of the interconnect's Maximum Transmission Unit (MTU).

(eg. 6 IP fragments for an Ethernet) There are serious problems with IP fragmentation, such as the need to resend the entire datagram if any one fragment is lost in transit, making this a poor transport mechanism for any but the most reliable network interconnects. [Kent87b] Since the 4.3BSD Reno implementation is transport layer independent, an experimental evaluation of performance over other transport mechanisms seemed appropriate.

Table #1			
Read Rate, large file (Kbytes/sec)			
Config.	#1	#2	#3
Transp.	LAN	80Mbit	56Kbit
UDP rto=A+4D	202	154	6.21
UDP rto=1sec	198	117	1.77
TCP	177	106	6.38



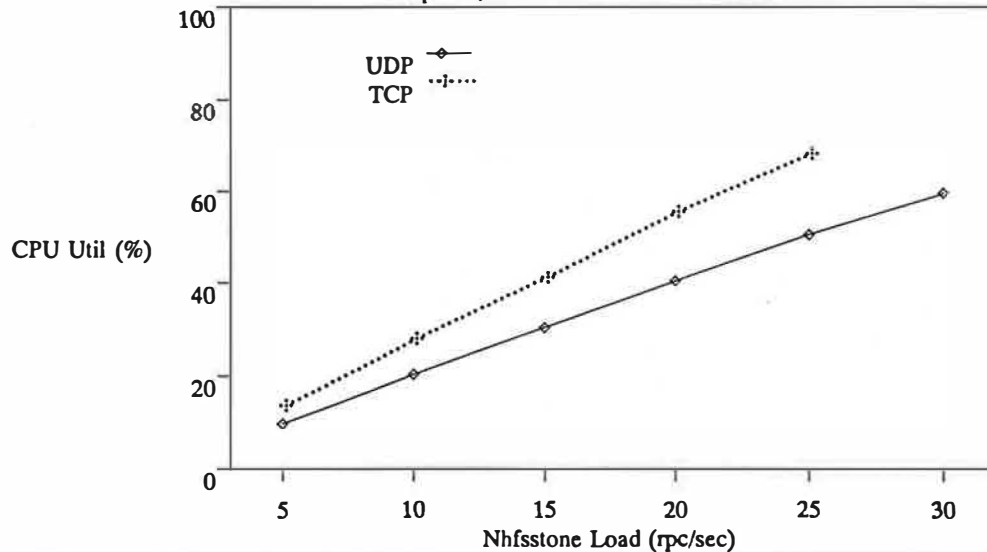
The first alternate transport mechanism was a reliable virtual circuit (TCP) protocol with dynamic RTO estimation and congestion control. [Jacobson88a] Although others, [Chesson87] had suggested that CPU overheads might be excessive, it was felt that the advantages of reliable transport with congestion control might outweigh the increased CPU overheads when using congested internetwork connections. Early informal observations indicated that TCP ran well enough and led to further interest in the next alternative.

The other alternate transport mechanism used UDP, but with dynamic RTO estimation and a congestion window on outstanding requests modelled after that of TCP. The advantage of this approach over TCP is that it does not *break* the NFS protocol and works with existing NFS servers. Trace data of round trip time (RTT) for the NFS RPCs indicated

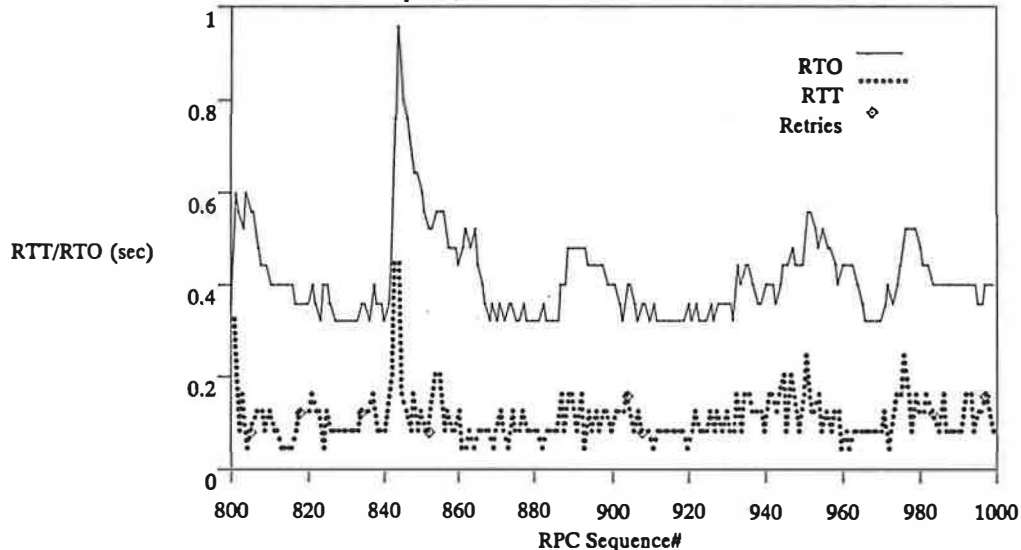
that different RPCs had vastly different RTTs and that the variance of *big* RPCs (Read, Write, Readdir) was higher than that of the *small* RPCs. (Getattr, Lookup) As such, it was decided to do separate RTO estimation on the four most frequent RPCs, **Read, Write, Getattr** and **Lookup** and use the constant value provided by the mount for the others. Since the *others* occur infrequently, it was felt that dynamic RTO estimation was impractical. Also, since most of these other RPCs are nonidempotent [Juszczak89], a conservative RTO is desired to minimize the risk of redoing the RPC. This design was somewhat similar to [Nowicki89] who used three timers and an overall estimation.⁴

⁴My implementation was actually based on work done by Tom Talpey of the OSF. I was not aware of the work done by [Nowicki89] until later. It was not obvious to me what Nowicki meant by *overall estimation*.

Graph #6, Read mix Server CPU Utilization



Graph #7, Read RPC RTT/RTO Trace across 80Mbit



The criteria for initial testing of these changes to UDP transport was that there should be no significant negative impacts when compared with the old UDP transport running on a single LAN. Early test runs showed that the retry rate for read RPCs was 2-4 times that of old UDP.

Two changes were made to bring the retry rate down:

- The calculation of RTO for the *big* RPCs (Read, Write, Readdir) was changed from "A+2D" to "A+4D" to allow for the large variances.⁵
- The RTO was recalculated on every NFS clock tick instead of at request transmission time, so that the most current values of A and D were

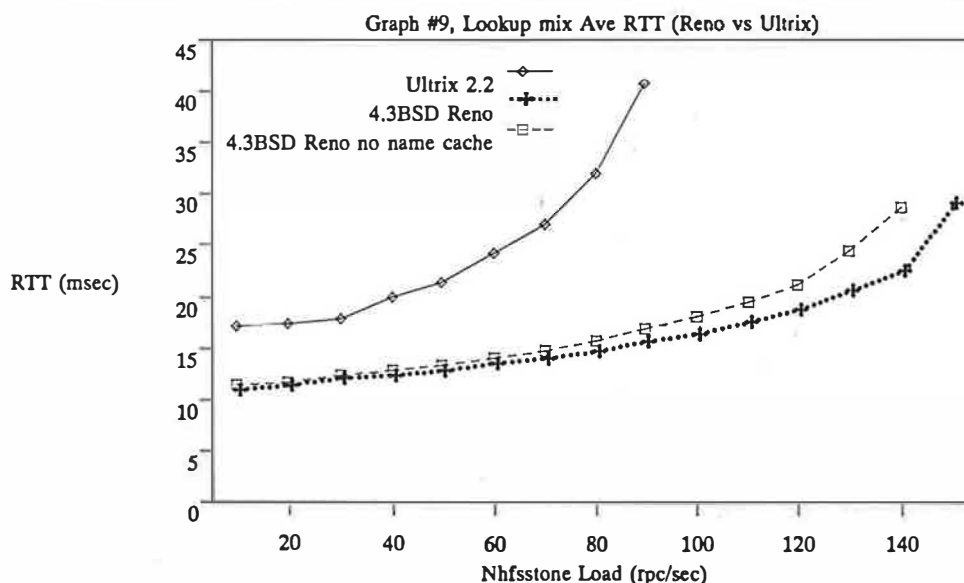
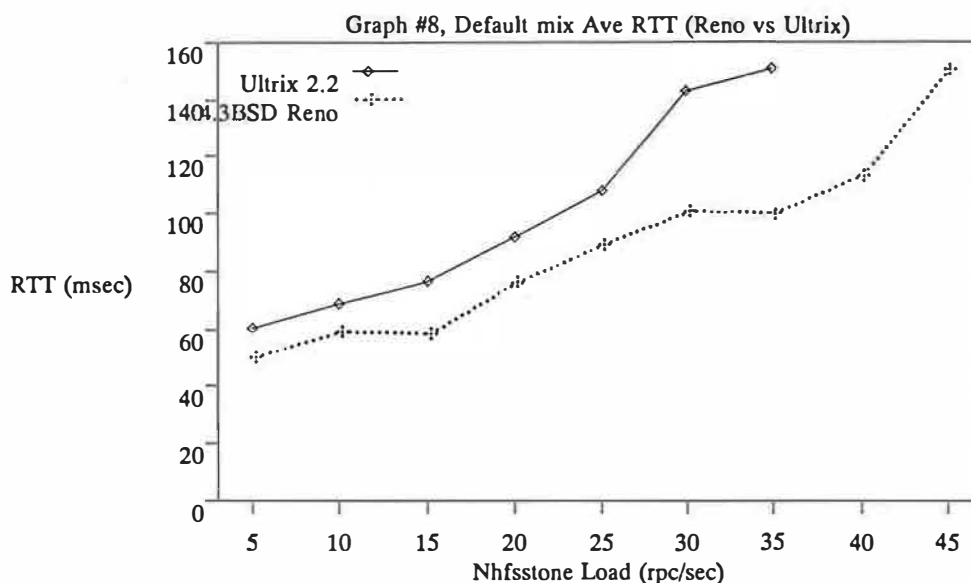
used.

It was also found that "slow start" impacted performance and had to be removed from the code. As a result, the congestion window on the number of outstanding RPCs is simply incremented by one for each RTT upon reception of an RPC reply and divided by two upon a retransmit timeout.

The experiment consisted of running an NFS RPC load between a client and server interconnected in three ways:

1. Both machines on the same uncongested Ethernet
2. Machines on two Ethernets interconnected by an 80Mbit/sec token ring and two IP routers.
3. Machines on two Ethernets interconnected by an 80Mbit/sec token ring, a 56Kbs point to point link and three IP routers.

⁵A is the estimated mean and D the estimated mean deviation of RTT



The NFS load was generated by the Nfhfsstone [Legato89] benchmark using two different load mixes: a 100% lookup RPC and a 50/50 lookup/read RPC. Since the object here was to measure the effects of different transport mechanisms, all that was required were *big* and *small* RPCs. Any RPCs that modify the underlying file system were avoided so that the subtree would remain stable and not require reloading between each test run. The 50/50 read/lookup load mix was selected since these are the most frequent *big* and *small* RPC's plus the fact that Nfhfsstone requires a high percentage of lookup RPCs to function well. The 100% lookup load mix was chosen to allow factoring out of the effect of lookups on the above. Each point in graphs #1-5 represent a test of 30min, to avoid momentary variations caused by other network loads. There were two runs done for each of the (*transport*, *internetwork-configuration*) tuples and each of these is represented by a line on one of the graphs. Since these tests were run across production networks during off peak hours, the other network loads were realistic but were not controlled nor reproducible. As such, it is probably the shape of the curves that is more relevant than the RTTs of individual data points. In the case of the 56Kbps link, after hours involved almost no other loads.

Graph #6 compares the server CPU overhead of UDP and TCP for an Nfhfsstone default RPC mix and Graph #7 is a sample trace of RTT and RTO equal A+4D for read RPCs.

Contrasting Graph #1 with #3 and #5, Graph #2 with #4 and examining Table #1, a variety of observations can be made:

- Graphs #1-2 - When both the client and server were on the same LAN, the method used to set RTO for UDP is not relevant. The RTTs for TCP are higher by a fixed amount of approximately 7msec for

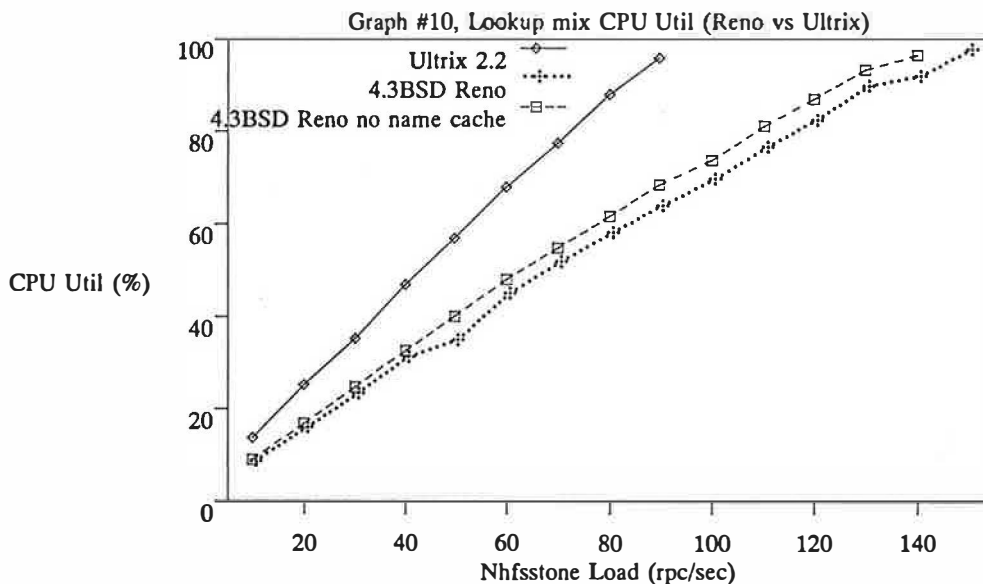
lookups and 10msec for the read mix until the server is under heavy load. For the read mix, much of this increased delay can be attributed directly to the higher CPU overhead associated with TCP. (7msec/rpc)⁶ As for lookups, the increase in CPU overhead is only 1msec/rpc for TCP, so there must be some other factor introducing real time delay. (Possibly more packets through the DEQNA Ethernet interface, which is *real slow*)

- Graphs #3-4 - When the client and server were interconnected through the 80Mbit token ring and gateways, the differences start to become apparent. The TCP curves are almost identical, indicating a high degree of stability. The curves for UDP with dynamic RTO estimation are somewhat more variable than TCP but with equal or better average RTTs, due to the lower CPU overheads. However, the curves for UDP with a fixed 1sec RTO are more erratic, due to the long delays before retransmits. At first glance, this would suggest that 1sec was too large, but examination of RTT trace data had peaks for Read RPCs at close to 1sec, which suggests that lowering the constant would not be advisable. The read rates for UDP with fixed RTO and TCP are almost the same, suggesting that the gains resulting from congestion control are cancelled out by the delay introduced by the higher CPU overhead. In this case, the simple congestion control added to UDP has improved read rate by about 30% over the other transport methods.

- When running across the 56Kbps link, the tests could only be run for the lookup mix.⁷ As graph #5⁸

⁶The test machines were 0.9MIPS MicroVAXIIs and as such a small amount of processing takes several msec. (Also See Graph #6)

⁷The upper bound on the number of 8Kbyte reads over a 7Kbyte link is < 1/sec.



and Table #1 indicate, UDP with a fixed 1sec RTO did not perform as well as either of the others. In this case, TCP performed consistently well and UDP with dynamic RTO estimation and congestion avoidance was often equal to TCP, but at times became unstable. The advantage of providing congestion control for this kind of network interconnect becomes apparent when you look at the read rates in Table #1. The read rates for TCP and UDP with dynamic RTO and congestion control are over three times that of UDP with fixed RTO.

There is another aspect of UDP transport for NFS and that is the choice of read/write data size. All of the above tests were run with the default 8Kbytes, but there are situations where decreasing the read/write size might improve performance. The difference in read rate between the two versions of UDP transport across the 56Kbyte link suggests that a congestion avoidance scheme may be sufficient for most situations, so that this is not normally required. Decreasing the read/write size increases the number of RPCs and as such should be considered as a last ditch action when all else fails. Since the trick here is to avoid IP fragment loss, it may be possible to adjust the size dynamically, based on the IP fragment drop rate. (This has not yet been tried, but is an area for further work.)⁹

5. Client Side Caching Issues

The 4.3BSD Reno NFS implementation uses several caching mechanisms that are believed to be somewhat different from those of the Sun NFS reference port. The 4.3BSD Reno VFS¹⁰ layer buffer cache is used by the NFS client to cache regular file blocks, directory blocks and symbolic links. There are references to these cached blocks hanging directly off of the vnodes. For writing of partial buffers there is no need to pre-read the blocks from the server, since there are additional fields in the buf¹¹ structure for keeping track of the "dirty" region within the buffer. File attributes are cached in the associated vnode¹² structure and there is also a VFS layer name lookup cache in 4.3BSD Reno. Any performance gains that could be related to differences in the caching mechanisms could suggest future work related to caching mechanisms.

⁸For one of the UDP rto=A+4D runs, the Ave RTT for 5rpc/sec was 721msec and therefore off the graph.

⁹[Nowicki89] describes some difficulties w.r.t. dynamically adjusting read/write size, but does not explain how they resolved the problems.

¹⁰VFS refers to the Virtual File System described in [Karels86].

¹¹buf is the Berkeley Unix structure for handling block I/O buffers.

¹²vnode is the structure in Berkeley Unix for a file object. See [Karels86]

An experimental mount flag that disables all of the NFS cache consistency mechanisms was implemented. Although operating NFS in this way is not practical in a production environment, it was done to allow determination of an optimistic bound on performance of a system with a cache consistency protocol.¹³ This is of interest, since distributed file systems such as Sprite have been observed to outperform NFS.¹⁴ [Nelson88] Another issue related to caching is, what is the best write policy?

The performance of a distributed file system implementation is closely coupled with network interface, disk I/O subsystem and processor performance. As such, use of identical hardware is required to isolate hardware related performance effects. A comparison with an implementation based on the Sun reference port was performed by benchmarking a MicroVAXII running both 4.3BSD Reno and Ultrix Version2.2. With these two systems as servers, Nfhstone loads were run on them from a client on the same LAN.

Graph #7 showed significant differences between 4.3BSD Reno and Ultrix. In order to isolate the basis of the differences, I ran further tests with 100% lookup and 50/50 read/lookup load mixes. The most significant difference was the lookup RPC performance, as seen in Graphs #8-9. An obvious explanation for the difference was the VFS name lookup cache on the 4.3BSD Reno server. However, disabling this cache only reduced the performance of 4.3BSD Reno by a small fraction of the difference observed compared to Ultrix. A possible explanation for the remainder of the difference is that on 4.3BSD Reno, the directory blocks in the server's buffer cache are chained directly off of the vnodes, reducing the CPU overhead for buffer cache searches.

Table #2			
Mod Andrew Bench	MicroVAXII client (sec)		
OS/Phase	I-IV	V	
Reno	145	1253	
Reno-TCP	143	1265	
Reno-nopush	132	1208	
Ultrix2.2	184	1183	

For client side testing, the Modified Andrew Benchmark [Osterhout90] was used with both systems mounting the same file system on the same

¹³Essentially a cache consistency protocol without overheads.

¹⁴Srinivasan et al were not able to achieve the performance gains of Sprite by adding a sprite like cache consistency protocol to NFS. They believed that a major reason for this was the large number of lookup RPCs that predominated. Since 4.3BSD Reno's name lookup cache reduces the number of lookup RPCs significantly, better performance improvements might be expected.

server. Since almost any *real work* is CPU bound on a MicroVAXII, the RPC counts in Table #3 are of more interest than the running times.

Table #3			
Mod Andrew Bench MicroVAXII client (RPC counts)			
RPC	Reno	Reno-noconsist	Ultrix2.2
Getattr	822	780	877
Setattr	22	22	22
Read	1050	619	691
Write	501	340	703
Lookup	872	918	1782
Readdir	146	144	150
Other	127	128	127
Total	3540	2951	4352

The biggest differences between 4.3BSD Reno and Ultrix were the number of lookups and the number of reads. The VFS name lookup cache on 4.3BSD Reno has reduced the number of lookup RPCs by 50%. This is significant, since lookup RPCs are usually the largest percentage of RPCs observed on production servers. The number of write RPCs was reduced by over 50% by disabling cache consistency. This implies a big reduction in server load, since every write RPC requires 1-3 disk writes on the server. The number of read RPCs for 4.3BSD Reno was 50% higher than Ultrix, and this can be traced to the fact that the 4.3BSD Reno NFS pushes all "dirty" blocks to the server before it starts reading a file. The argument for this is that after doing a write RPC, the modify time has changed, but the client cannot tell whether this modify was due to changes it made or to writes just done by other clients to the same file. It appears that Ultrix assumes that other clients are not writing to the same file at the same time and therefore regards data in the cache as still valid. As such, it may be worthwhile to rethink the above consistency criteria for 4.3BSD Reno.

The Modified Andrew Benchmark was also run on a DECstation 3100 against both servers to see what effect the server differences would have on real work. The results in Table #4 show a difference of 20-30% between the two servers.

Table #4			
Mod Andrew Bench DS3100 client (sec)			
OS/Phase	I-IV	V	
Reno	88	180	
Ultrix2.2	123	226	

To look at the effects of different write policies, the Create-Delete benchmark [Ousterhout90] was run with and without cache consistency. The tests were run with zero, four and sixteen biods,¹⁵ to simulate different levels of asynchronous I/O

¹⁵biod is a daemon that does asynchronous I/O for client NFS

concurrency. With no biods running, the write policy becomes *write through*.

Table #5			
Create-Delete Bench 4.3BSD Reno MicroVAXII (msec)			
Config	No data	10Kbytes	100Kbytes
Local	120	216	1170
write thru	210	475	2401
async,4biod	216	470	1940
async,16biod	210	464	2094
delay wrt.	216	468	2230
no consist	218	244	329

When maintaining close/open consistency by pushing writes on close, the only time that selection of write policy is significant is for large files. For the 100Kbyte file, it was observed that an *asynchronous write* policy was about 20% faster than *write through* or *delayed write*. However, there is a big improvement if you do not **push writes on close** due to the fact that there is usually no need for the write system call to block waiting for write RPCs to complete. Also, the number of write RPCs is dramatically higher for asynchronous writes than for the delayed write without push on close, (Table #3) suggesting that there is a good argument for this approach based on reduced server load. (Also see [Nelson88]) Note however, that to do this for a production environment would require the addition of some sort of cache consistency protocol to NFS.

Conclusions

The performance of an NFS implementation is influenced by caching performance for the client and caching plus CPU overhead for the server. Most current NFS servers have observed loads that are lookup RPC dominant. A good lookup name cache on the client can reduce the lookup RPC load significantly, causing the performance of the read/write RPCs to become more dominant. The read/write RPC performance of a server can be significantly improved by minimization of memory to memory copying and tuning of the low level network interface handling code.

Two of the major limitations of NFS are actually a result of the implementation of Sun RPC on UDP transport. The *at least once* semantics of these RPC's can result in faulty behaviour on a heavily loaded server, due to the repetition of non-idempotent RPCs. Also, the simple timeout/retransmit scheme used to achieve reliability is inadequate for all but the most reliable client/server interconnects. Serious degradation of performance has been observed across even a single IP gateway. Early evidence suggests that UDP transport can be improved by dynamic RTO estimation

¹⁵don't push writes on close is the major effect of disabling cache consistency

and a congestion window modelled after that used by TCP. It has also been found that TCP performs fairly well as an NFS transport mechanism, with an increase in CPU overhead of about 20% over UDP.

A cache consistency protocol would reduce the number of write RPCs by at least half.

Future Directions

As CPU speed increases, real work becomes less CPU bound and more sensitive to I/O performance [Ousterhout90]. As such, a performance evaluation of the client side running on a 20 MIPS workstation could yield further insight into appropriate client side caching mechanisms. In particular, with a reduction in lookup RPC rate due to a name lookup cache, it may be possible to achieve higher performance gains from a Sprite like cache consistency protocol than was observed by Srinivasan et al. [Srinivasan89] The experimental mount option for *don't do cache consistency* permits determination of an optimistic bound on the performance gains of such a Sprite like cache consistency protocol but does not solve the problem. A cache consistency protocol that is crash and network partition tolerant is still needed. A question here is whether full cache coherency is required or simply a mechanism for doing a *delayed write without push on close* policy safely.¹⁶

More work needs to be done on good transport mechanisms for RPC's. An improved timeout/retransmit scheme for UDP would be a first step, since there are so many NFS/UDP servers out there today. However, in the future I believe that UDP needs to be replaced as a transport mechanism for RPC's.

It would be desirable to construct some sort of experimental test bed to explore performance issues related to many gateway hops and *long fat pipes*. [Jacobson88b] Such a testbed could be used for experimentation with transport mechanisms and caching techniques better suited to large delay paths. To achieve good performance in these internetworks, the number of times that an I/O system call blocks for an RPC reply¹⁷ must be minimized. This would be achieved in part by a cache consistency protocol. However, I think that you must also do more cache preloading. There are many possibilities here. For reads, you might either increase the size of the read RPCs or the level of read-aheads¹⁸ or both, so that most read system calls find the data already in the

¹⁶This is not meant to imply that a *delayed write without push on close* protocol that retains close/open consistency criteria, handles *disk full* errors and server crashes is simple.

¹⁷[Ousterhout90] refers to this as decoupling I/O.

¹⁸Normally Unix does a read-ahead of 1 block. By increasing the level of read-ahead, I mean doing a read-ahead of the next 2-4 blocks.

local cache. I think that you also need a way of doing many name lookups per RPC, possibly by adding a *readdir_and_lookup_files* RPC to the protocol.

Acknowledgements

I am indebted to the people at the Computer Systems Research Group of the University of California Berkeley for their friendly poking and prodding over the past couple of years. Professors Jim Linders and Tom Wilson here at Guelph provided much needed guidance and support. The OSF has provided both moral and financial support for this activity and DEC helped immensely by providing a 50% equipment grant. I must also thank the personnel of Computing Services and Communications Services on the Guelph campus for permitting the use of their facilities for benchmarking.

References

- [Cheriton86] David R. Cheriton, VMTP: A Transport Protocol for the Next Generation of Communication Systems, In *Proc. SIGCOMM 86 Symposium on Communications Architectures and Protocols*, pg. 406-415, Stowe VT, August 1986.
- [Chesson87] G. Chesson, Protocol Engine Design, In *Proc. Summer 1987 USENIX Conference*, Phoenix, AZ, June 1987.
- [Jacobson88a] Van Jacobson, Congestion Avoidance and Control, In *Proc. SIGCOMM 88 Symposium on Communication Architectures and Protocols*, pg. 314-329, Stanford, CA, August 1988.
- [Jacobson88b] Van Jacobson and R. Braden, *TCP Extensions for Long-Delay Paths*, ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, October 1988, RFC-1072.
- [Jacobson89] Van Jacobson, Sun NFS Performance Problems, *Private Communication*, November, 1989.
- [Juszczak89] Chet Juszczak, Improving the Performance and Correctness of an NFS Server, In *Proc. Winter 1989 USENIX Conference*, pg. 53-63, San Diego, CA, January 1989.
- [Karels86] Michael J. Karels and Marshall Kirk McKusick, Toward a Compatible Filesystem Interface, In *Proc. EUUG Conference*, September 1986.
- [Keith90] Bruce E. Keith, Perspectives on NFS File Server Performance Characterization, In *Proc. Summer 1990 USENIX Conference*, pg. 267-277, Anaheim, CA, June 1990.
- [Kent87a] Christopher. A. Kent, *Cache Coherence in Distributed Systems*, Research Report 87/4, Digital Equipment Corporation Western Research Laboratory, April 1987.

- [Kent87b] Christopher A. Kent and Jeffrey C. Mogul, Fragmentation Considered Harmful, In *Proc. SIGCOMM 87 Workshop on Frontiers in Computer Communications Technology*, pg. 390-401, Stowe, VT, August 1987.
- [Nelson88] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, Caching in the Sprite Network File System, *ACM Transactions on Computer Systems* (6)1 pg. 134-154, February 1988.
- [Nhfsstone] Nhfsstone NFS load generating program, Legato Systems Inc., Palo Alto, CA, 94306.
- [Nowicki89] Bill Nowicki, Transport Issues in the Network File System, In *Computer Communication Review*, pg. 16-20, March 1989.
- [Ousterhout90] John K. Ousterhout, Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proc. Summer 1990 USENIX Conference*, pg. 247-256, Anaheim, CA, June 1990.
- [Reid90] Jim Reid, N(e)FS: the Protocol is the Problem, In *Proc. Summer 1990 UKUUG Conference*, London, England, July 1990.
- [Sandberg85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon, Design and Implementation of the Sun Network filesystem, In *Proc. Summer 1985 USENIX Conference*, pages 119-130, Portland, OR, June 1985.
- [Srinivasan89] V. Srinivasan and Jeffrey C. Mogul, *Spritely NFS: Implementation and Performance of Cache-Consistency Protocols*, Research Report 89/5, Digital Equipment Corporation Western Research Laboratory, May 1989.
- [RFC1094] Sun Microsystems Inc., *NFS: Network File System Protocol Specification*, ARPANET Working Group Requests for Comment, DDN Network Information Center, SRI International, Menlo Park, CA, March 1989, RFC-1094.

Rick Macklem is a systems supervisor for forty odd Unix boxes that support teaching and research in the Department of Computing and Information Science at the University of Guelph. Since his first introduction to Version 6 in 1977, Rick has become a kernel hack, having learned to avoid the more mundane aspects of system management like the plague. You can reach Rick electronically at rick@snowwhite.cis.uoguelph.ca and via surface mail at Department of Computing and Info. Sc., University of Guelph, Guelph, Ont. N1G 2W1, Canada.



Appendix: Experimental Details

All tests were performed on identical hardware, MicroVAXIIs' with RD53 disks and a DEQNA ethernet interface attached to either a lightly loaded Ethernet or the internetworks described in Section 4. Although not representative of current hardware, these systems demonstrate relatively well balanced performance (ie. slow CPU, slow disks and a slow network interface) and were the only systems available that would run both 4.3BSD Reno and a vendor implementation of NFS based on the Sun reference port. The emphasis was placed on the four RPCs *getattr*, *lookup*, *read* and *write*, since these make up a majority of most NFS RPC workloads. For client side benchmarking, the same server was always used. For the Modified Andrew Benchmark, the DECstation 3100 was selected as it has a sufficiently fast processor so that real work, such as C compilation, is not entirely CPU bound. For comparisons between 4.3BSD Reno and the vendor kernel (*Ultrix Version 2.2*), the kernels were configured with identically sized buffer caches and file systems.

The percentage of idle CPU as reported by *ios-tat(1)* was observed to be erratic during early test runs. The cause of this was found to be a hardware constraint of the MicroVAXII, which masks off clock interrupts during peripheral interrupts. To avoid this problem, all kernels were patched with a counter inside the idle loop to allow for an accurate measure of CPU utilization. This is a particularly handy bit of instrumentation, since it does not have any adverse effect on real performance due to the fact that the instrumentation overhead is only incurred when the CPU is idle.

Two caveats were identified in the Nhfsstone¹⁹ server characterization benchmark as follows:

- 1) The Nhfsstone benchmark uses long file names to defeat client name caching, but this can also defeat server name caching. This will tend to bias against servers with good lookup name caches. To determine the extent of this problem, the lookup benchmark was run against a server with and without name caching enabled.²⁰
- 2) The Nhfsstone benchmark chooses a file at random and then performs a random operation on it in proportion to its load mix. Since most load mixes have a small proportion of writes (8% is the default), starting with empty test directories causes most files to remain empty during the test interval. This implies that most reads are performed on empty files and biases the results against a server with good read performance. Further, as testing continues, more files are written reducing the number of empty files. This

²⁰4.3BSD Reno name caches file names up to 31 characters, which is longer than the names used by the Nhfsstone benchmark.

results in the average RTT increasing over time, due to the fact that fewer of the reads are of empty files. To avoid this side effect, the subtree was preloaded with an identical set of files before each test.

SunOS Multi-thread Architecture

M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks – Sun Microsystems Inc.

ABSTRACT

We describe a model for multiple threads of control within a single UNIX process. The main goals are to provide extremely lightweight threads and to rationalize and extend the UNIX Application Programming Interface for a multi-threaded environment. The threads are intended to be sufficiently lightweight so that there can be thousands present and that synchronization and context switching can be accomplished rapidly without entering the kernel. These goals are achieved by providing lightweight user-level threads that are multiplexed on top of kernel-supported threads of control. This architecture allows the programmer to separate logical (program) concurrency from the required real concurrency, which is relatively costly, and to control both within a single programming model.

Introduction

The reasons for supporting multiple threads of control in SunOS fall into two categories, those motivated by multiprocessor hardware and those motivated by application concurrency. It is possible to exploit multiprocessors to varying degrees depending on how much the uniprocessor software base is modified. In the simplest case, only separate user processes can run on the additional processors; the applications are unchanged. To allow a single application to use multiple processors (e.g. array processing workload), the application must be restructured.

The second category of reasons for multiple threads of control is application concurrency. Many applications are best structured as several independent computations. A database system may have many user interactions in progress while at the same time performing several file and network operations. A window system can treat each widget as a separate entity. A network server may indirectly need its own service (and therefore another thread of control) to handle requests. In each case, although it is possible to write the software as one thread of control moving from request to request, the code may be simplified by writing each request as a separate sequence, and letting the language, library, and operating system handle the interleaving of the different operations.

These examples are not intended to be exhaustive, but they indicate the opportunities to exploit powerful hardware and build complex applications and services with this technology. The examples show that the user model for multiple threads of control must support a variety of applications and environments. The architecture should, where possible, use current programming paradigms and preserve software compatibility.

As is true of many system services today, the programmer's view of the multiple threads of control service is not always identical to what the kernel

implements. The software view is created by a combination of the kernel, run-time libraries, and the compilation system. This approach increases the portability of applications and systems, by hiding some details of the implementation, while providing better performance, by allowing library code to do some work without involving the kernel.

The remainder of this paper is divided into five sections. The first section gives an overview of the architecture and introduces our terminology. The second section discusses our design goals and principles. The third section gives additional details of operation and interfaces and how the UNIX process model is reinterpreted in the new environment. The fourth section gives some performance data and operational experience. The last section compares this architecture with others.

The terminology of multiprocessor and multi-threaded computation is unfortunately not universally agreed upon. We have chosen terms that are most common and have tried to be consistent, but the reader is warned that some people use these words with other meanings. Examples of other models can be found in the last section of this paper.

Multi-Threading Architecture Overview

The multi-threaded programming model has two levels. The most important level is the thread interface, which defines most aspects of the programming model. That is, programmers write programs using threads. The second level is the lightweight process (LWP) which is defined by the services the operating system must provide. After describing each level, we explain why both levels are essential.

Threads

A traditional UNIX process has a single thread of control. A thread of control, or more simply a thread, is a sequence of instructions being executed in a program. A thread has a program counter (PC)

and a stack to keep track of local variables and return addresses. A multi-threaded UNIX process is no longer a thread of control in itself, instead it is associated with one or more threads. Threads execute independently. There is in general no way to predict how the instructions of different threads are interleaved, though they have execution priorities that can influence the relative speed of execution. In general, the number or identities of threads that an application process chooses to apply to a problem are invisible from outside the process. Threads can be viewed as execution resources that may be applied to solving the problem at hand.

Threads share the process instructions and most of its data. A change in shared data by one thread can be seen by the other threads in the process. Threads also share most of the operating system state of a process. Each sees the same open files. For example, if one thread opens a file, another thread can read it. Because threads share so much of the process state, threads can affect each other in sometimes surprising ways. Programming with threads requires more care and discipline than ordinary programming because there is no system-enforced protection between threads.

Each thread may make arbitrary system calls and interact with other processes in the usual ways. Some operations affect all the threads in a process. For example, if one thread calls `exit()`, all threads are destroyed. Other UNIX system services have new interpretations; e.g. a floating-point overflow trap applies to a particular thread, not the whole program.

The architecture provides a variety of synchronization facilities to allow threads to cooperate in accessing shared data. The synchronization facilities include mutual exclusion (mutex) locks, condition variables and semaphores. For example, a thread that wants to update a variable might block waiting for a mutual exclusion lock held by another thread that is already updating it. To support different frequencies of interaction and different degrees of concurrency, several synchronization mechanisms with different semantics are provided.

As shown in Figure 1, threads in different processes can synchronize with each other via synchronization variables placed in shared memory,

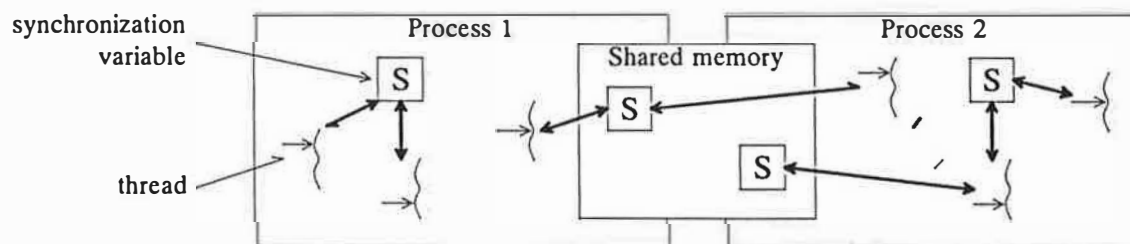
even though the threads in different processes are generally invisible to each other. Synchronization variables can also be placed in files and have lifetimes beyond that of the creating process. For example, a file can be created that contains data base records. Each record can contain a mutual exclusion lock variable that controls access to the associated record. A process can map the file and a thread within it can obtain the lock associated with a particular record that is to be modified. When the modification is complete the thread can release the lock and unmap the file. Once the lock has been acquired, if any thread within any process mapping the file attempts to acquire the lock that thread will block until the lock is released.

Lightweight processes

Threads are an appropriate paradigm for most programs that wish to exploit parallel hardware or express concurrent program structure. For those situations that require more control over how the program is mapped onto parallel hardware, and to optimize the costs of concurrent execution and synchronization, a second interface is defined.

In the SunOS multi-thread architecture, a UNIX process consists mainly of an address space and a set of lightweight processes (LWPs)¹ that share that address space. Each LWP can be thought of as a virtual CPU which is available for executing code or system calls. Each LWP is separately dispatched by the kernel, may perform independent system calls, incur independent page faults, and may run in parallel on a multiprocessor. All the LWPs in the system are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority.

Threads are implemented using LWPs. Threads are actually represented by data structures in the address space of a program. LWPs within a process execute threads as shown in Figure 2. An LWP chooses a thread to run by locating the thread state



¹The LWPs in this document are fundamentally different than the LWP library in SunOS 4.0. Lack of imagination and a desire to conform to generally accepted terminology lead us to use the same name.

in process memory (a). After loading the registers and assuming the identity of the thread, the LWP executes the thread's instructions (b). If the thread cannot continue, or if other threads should be run, the LWP saves the state of the thread back in memory (c). The LWP can now select another thread to run (d).

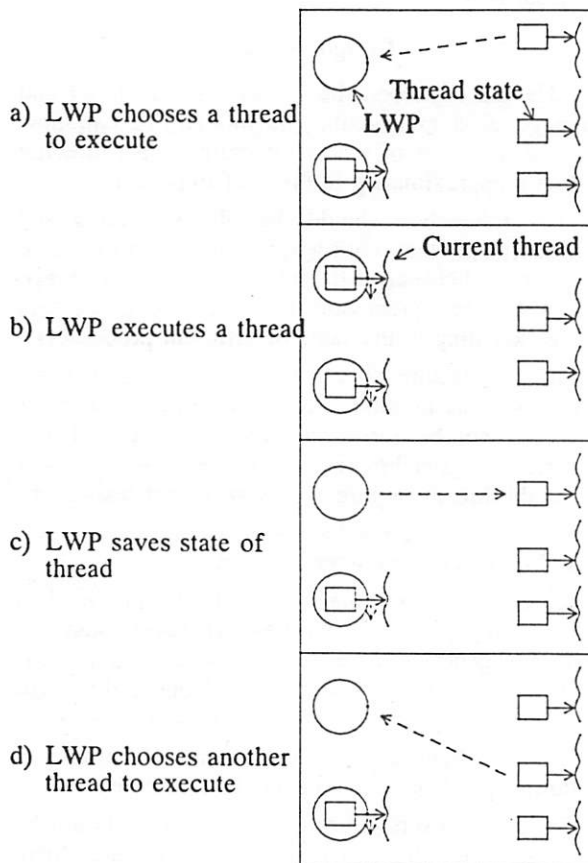


Figure 2: LWP running threads

When a thread needs to access a system service by performing a kernel call, taking a page fault, or to interact with threads in other processes, it does so using the LWP that is executing it. The thread needing the system service remains bound to the LWP executing it until the system call is completed. If a thread needs to interact with other threads in the same process, it can do so without involving the operating system. As Figure 2 shows, switching from one thread to another occurs without the kernel knowing it. Much as the UNIX `stdio` library routines (such as `fopen()` and `fread()`) are implemented using the UNIX system calls (`open()` and `read()`), the thread interface is implemented using the LWP interface, and for many of the same reasons.

An LWP may also have some capabilities that are not exported directly to threads, such as a special scheduling class. A programmer can take advantage of these capabilities while still retaining use of all

the thread interfaces and capabilities (e.g. synchronization) by specifying that the thread is to remain permanently bound to an LWP.

Threads are the primary interface for application parallelism. Few multi-threaded programs will use the LWP interface directly, but it is sometimes important to know that it is there. Some languages define concurrency mechanisms that are different from threads. An example is a Fortran compiler that provides loop level parallelism. In such cases, the language library may implement its own notion of concurrency using LWPs. Most programmers can program using the threads interface and let the library take care of mapping threads onto the kernel primitives. The decision of how many LWPs should be created to run the threads can be left to the library, or may be specified by the programmer.

Why have both threads and LWPs?

One might wonder why it is necessary to have two interfaces that are so similar. The multi-threaded architecture must meet a variety of different expectations. Some programs have large amounts of logical parallelism, such as a window system that provides each widget with one input handler and one output handler. Other programs need to map their parallel computation onto the actual number of processors available. In both cases, programs want to easily have complete access to the system services.

Threads are implemented by the library and are not known to the kernel. Thus, threads may be created, destroyed, blocked, activated, etc., without involving the kernel. LWPs are implemented by the kernel. If a thread wants to read from a file, the kernel needs to be able to switch to other processing when the LWP blocks in the file system code waiting for the I/O to finish. The kernel has to preserve the state of the read operation and continue it when the I/O interrupt arrives. However, if each thread were always known to the kernel, it would have to allocate kernel data structures for each one and get involved in context switching threads even though most thread interactions involve threads in the same process. In other words, kernel-supported parallelism (LWPs) is relatively expensive compared to threads. Having all threads supported directly by the kernel would cause applications such as the window system to be much less efficient. Although the window system may be best expressed as a large number of threads, only a few of the threads ever need to be active (i.e. require kernel resources, other than virtual memory) at the same instant.

Sometimes having more threads than LWPs is a disadvantage. A parallel array computation divides the rows of its arrays among different threads. If there is one LWP per processor, but multiple threads per LWP, each processor would spend overhead switching between threads. It would be better to

know that there is one thread per LWP, divide the rows among a smaller number of threads, and reduce the number of thread switches. By specifying that each thread is permanently bound to its own LWP, a programmer can write thread code that is really LWP code, much like locking down pages turns virtual memory into real memory.

A mixture of threads that are permanently bound to LWPs and unbound threads is also appropriate for some applications. An example of this would be some real-time applications that want some threads to have system-wide priority and real-time scheduling, while other threads can attend to background computations.

By defining both levels of interface in the architecture, we make clear the distinction between what the programmer sees and what the kernel provides. Most programmers program using threads and do not think about LWPs. When it is appropriate to optimize the behavior of the program, the programmer has the ability to tune the relationship between threads and LWPs. This allows programmers to structure their application assuming extremely lightweight threads while bringing the appropriate degree of kernel-supported concurrency to bear on the computation. To some degree, a threads programmer can think of LWPs used by the application as the degree of real concurrency that the application requires.

Summary

Figure 3 shows all of the pieces in one diagram. The assignment of threads to LWPs is either controlled by the threads package or is specified by the programmer. The kernel sees LWPs and may schedule these on the available processors.

Process 1 is the traditional UNIX process with a single thread attached to a single LWP. Process 2 has threads multiplexed on a single LWP as in typical coroutine packages, such as SunOS 4.0 liblwp. Process 3 through 5 depict new capabilities of the SunOS multi-thread architecture. Process 3 has several threads multiplexed on a lesser number of LWPs. Process 4 has its threads permanently bound

to LWPs. Process 5 shows all the possibilities; a group of threads multiplexed on a group of LWPs, while having threads bound to LWPs. In addition, the process has asked the system to bind one of its LWPs to a CPU. Note that the bound and unbound threads can still synchronize with each other both within the same process and between processes in the usual way.

Design Goals

Having described the overall thread model and language used to describe the model, we can now describe the goals of the architecture. The following goals are approximately in order of importance.

- The architecture should describe structures and mechanisms that work among threads in the same program, between different programs (processes), and between processors (whether the processors are executing in the same or different processes).
- The architecture should support threads that are as cheap as possible. Threads within a program should not be forced to cross protection boundaries to synchronize or context switch, nor should threads require excessive kernel resources.
- The architecture must support both multiprocessor and uniprocessor implementations.
- All current UNIX semantics should be provided in user programs and libraries wherever possible. The degenerate case of a process being constructed of an address space and one lightweight process must provide complete UNIX semantics.
- Different lightweight processes should be able to do independent, simultaneous system calls.
- The mechanisms defined in the system should be simple and fundamental. For example, there should be a method of using threads that does not force the threads library to use `malloc()`. This prevents interference with other application or language run-time system memory allocators.

The following are not exactly goals, but are principles that were used to help design the architecture.

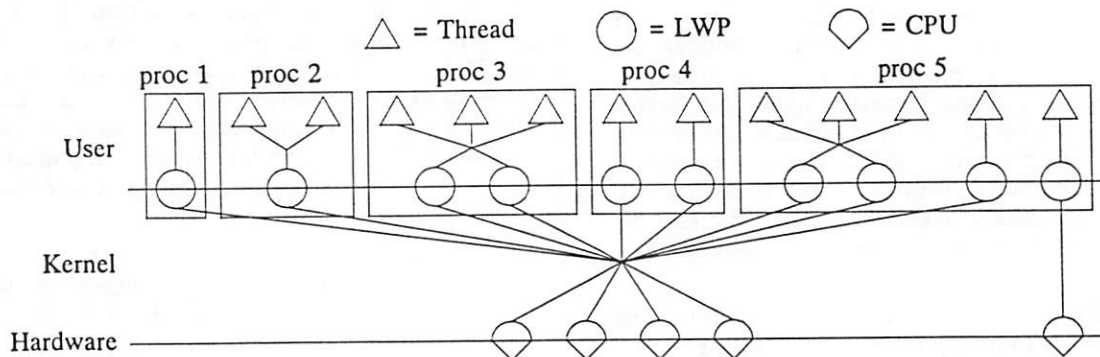


Figure 3: Multi-thread architecture examples

- Per-thread state must be kept to a minimum. Each additional piece of state above the minimum necessary must be justified so as not to add undue "weight" to a thread.
- An address space with one thread (and therefore one lightweight process) should behave like a standard UNIX process; the addition of a new thread (and possibly a lightweight process) that does not interact with the first thread should not change the behavior of the first thread.
- The opportunity should be provided for different implementations. For example, by allowing but not requiring threads to share the whole address space, by allowing but not requiring threads to be multiplexed on lightweight processes, and by allowing but not requiring synchronization primitives to be executed in user mode.
- Wherever possible, equivalent semantics to UNIX should be provided, even if that doesn't seem like the best way to implement the function. Alternative operations should be added to do things the "right" way.
- The process is the unit of work. Threads are resources of the process and are applied to the work of the process in much the same way as file descriptors. For example, threads in other processes are invisible.

Multi-threaded Operations

System calls

The base programmer interface for functions other than those relating to threads or multi-threading is the System V Interface Definition, Third Edition (SVID3). In general, most current UNIX system calls remain unchanged. The main difference is that system calls that block do so to the lightweight process and therefore to the thread that executes them. However programmers must understand that threads and LWP's share almost all the programmer visible process resources such as address space and file descriptor table. This can lead to several potential trouble spots:

- Because file descriptors are shared, if one thread closes a file, it is closed for all threads. Care must be taken with seeks before reads or writes, because another thread could change the seek position before the read or write (this is similar to what happens now when a parent and child process share a file descriptor).
- There is only one working directory for each process. If one thread changes the working directory, it is changed for all of them.
- There is only one set of user and group IDs for each process, so if one thread changes one of these, it is changed for all of them. Because these can change concurrently, the kernel must ensure that the values are sampled, atomically, only once

per system call.

- Multiple threads may manipulate the shared address space at the same time via `mmap()`, `brk()`, or `sbrk()`.
- Programs must not make assumptions about "the" stack, because there may be several.

Threads and lightweight processes

One lightweight process is created by the kernel when a program is started, and it starts executing the thread compiled as the main program. Additional threads are created by calls to the library specifying a procedure for the new thread to execute and a stack area for it to use.

Depending on the implementation, on the library, or on programmer supplied parameters, a thread may be associated with the same or different lightweight processes during its lifetime. There may be a one-to-one relationship between threads and lightweight processes, or one or more lightweight processes may be multiplexed by the thread library among a set of threads. Ordinarily, a thread cannot tell what the relationship between lightweight processes and threads is, although for performance reasons, or to avoid some deadlocks, a program may require there to be more or fewer lightweight processes.

When a thread executes a kernel call, it remains bound to the same lightweight process for the duration of the kernel call. If the kernel call blocks, that thread and its lightweight process remain blocked. Other lightweight processes may execute other threads in that program, including performing other kernel calls. The same principle applies to page faults.

There is no system-wide name space for threads or lightweight processes. Thus, for example, it is not possible to direct a signal to a particular lightweight process from outside a process or to know which lightweight process sent a particular message.

Thread state

The following state is unique to each thread:

- Thread ID
- Register state (including PC and stack pointer)
- Stack
- Signal mask
- Priority
- Thread-local storage

All other process state is shared by the threads within the process.

Thread-local storage

Threads have some private storage (in addition to the stack) called thread-local storage. Most variables in the program are shared among all the threads executing it, but each thread has its own copy of thread-local variables. Conceptually, thread-local storage is unshared, statically allocated data. The C library variable `errno` is a good example of a variable that should be placed in thread-local storage. This allows each thread to reference `errno` directly and it allows threads to interleave execution without fear of corrupting `errno` in other threads. Thread-local storage is potentially expensive to access, so it should be limited to the essentials, such as supporting older, non-reentrant interfaces.

It is implementation-dependent whether or not a thread is absolutely prevented from accessing another thread's stack or thread-local variables, but a correct thread must never attempt to do so.

Thread-local storage is obtained via a new `#pragma`, supported by the compiler and linker. The contents of thread-local storage are zeroed, initially; static initialization is not allowed. In C, thread-local storage for `errno` would be declared as follows:

```
#pragma unshared errno
extern int errno;
```

The size of thread-local storage is computed by the run-time linker at program start time by summing the thread-local storage requirements of the linked libraries. This prevents the exact size of thread-local storage from being part of the library interface. Once the size is computed it is not changed (e.g. by future dynamic linking in the process). This restriction prevents the size of thread-local storage from changing once a thread is started. Thus thread-local storage requirements are known at thread startup time and can be allocated as part of stack storage. More dynamic mechanisms (such as POSIX thread-specific data [POSIX 1990]) can be built using thread-local storage.

Thread synchronization

Threads synchronize with each other using facilities supplied by the implementation that present a standard set of semantics. The following synchronization types are supported:

- Mutual exclusion (mutex) locks
- Counting semaphores
- Condition variables
- Multiple readers, single writer locks

The architecture allows a range of implementations of each synchronization type to be supported. For example, mutual exclusion locks may be implemented as spin locks, sleep locks, or adaptive locks, etc.

These facilities use synchronization variables in memory. The variables may be statically allocated and/or at fixed addresses (within the alignment constraints of the variable). The programmer may choose the particular implementation variant of the synchronization semantic at the time the variable is initialized. If the variable is initialized to zero, a default implementation is used.

Synchronization variables may also be placed in memory that is shared between processes. The programmer can select an implementation variant of each synchronization type that allows the variable to synchronize threads in the processes sharing the variable. Synchronization primitives apply to the shared variable as part of the underlying mapped object. In other words, synchronization variables may be shared between processes even though they are mapped at different virtual addresses.

Synchronization variables that are not in shared memory are completely unknown to the kernel. Synchronization variables that are in shared memory or in files are also unknown to the kernel unless a thread is blocked on them. In the latter case the thread is temporarily bound to the LWP that is blocked by the kernel, as in a system call.

Signal handling

Each thread has its own signal mask. This permits a thread to block some signals while it uses state that is also modified by a signal handler. All threads in the same address space share the set of signal handlers, which are set up by `signal()` and its variants, as usual. If desired, it would be possible for a particular application to implement per-thread signal handlers using the per-process signal handlers. For example, the signal handler can use the ID of the thread handling the signal as an index into a table of per-thread handlers. If the threads library were to implement per-thread signal handlers it must decide on the correct semantics when several threads have different combinations of signal handlers, `SIG_IGN`, and `SIG_DFL`. In addition, all threads would be burdened with the handler state. For this reason, we felt that library support of per-thread signal handlers was overly complex and possibly confusing to the application programmer.

If a signal handler is marked `SIG_DFL` or `SIG_IGN` the action on receipt of the signal (exit, core dump, stop, continue, or ignore) affects all the threads in the receiving process.

Signals are divided into two categories: traps and interrupts. Traps (e.g. `SIGILL`, `SIGFPE`, `SIGSEGV`) are signals that are caused synchronously by the operation of a thread, and are handled only by the thread that caused them. Several threads in the same address space could conceivably generate and handle the same kind of trap simultaneously. Interrupts (e.g. `SIGINT`, `SIGIO`) are signals that are

caused asynchronously by something outside the process. An interrupt may be handled by any thread that has it enabled in its signal mask. If more than one thread is enabled to receive the interrupt, only one is chosen. Thus, several threads can be in the process of handling the same kind of signal simultaneously. If all threads mask a signal, it will pend on the process until a thread unmask that signal. As in single-threaded processes, the number of signals received by the process is less than or equal to the number sent.

For example, an application can enable several threads to handle a particular I/O interrupt. As each new interrupt comes in, another thread is chosen to handle the signal until all the enabled threads are active. New signals will then pend waiting for threads to complete processing and re-enable signal handling.

Threads may send signals to other threads within the process via a new interface; `thread_kill()`. In this case the signal behaves like a trap and can be handled only by the specified thread. The programmer may also send a signal to all the threads via `sigsend()`. A thread cannot send a signal to a specific thread in another process because threads in other processes are invisible.

Threads that are not bound to LWPs may not use alternate signal stacks. Adding alternate signal stacks to the unbound thread state was deemed too expensive to implement because this would require a system call to establish the alternate stack for each context switch of a thread requiring it. Threads bound to LWPs may use alternate stacks as this state is associated with each LWP.

Non-local goto

`setjmp()` and `longjmp()` work only within a particular thread. In particular, it is an error for a thread to `longjmp()` into another thread. Therefore, it is possible to `longjmp()` from a signal handler only when the `setjmp()` was executed by the thread that is handling the signal.

Thread interfaces

Most of the interfaces available to threads are those that are available to UNIX processes in single-threaded UNIX. As mentioned above, some of those interfaces have different implications in a multi-threaded environment, but the intent is to provide "UNIX semantics" as the ordinary programming model. This section describes some of the additional interfaces needed to create and manage threads.

The syntax of the interfaces is shown in Figure 4.

Thread creation

`thread_create()` creates a new thread. If `stack_addr` is not NULL, `stack_size` bytes of memory starting at `stack_addr` are used for the thread stack. In this case any thread-local storage is also placed on the stack so as not to interfere with stack growth. This allows a language run-time library to control thread storage without interference with its memory allocator. It is machine dependent whether the initial stack pointer is at higher or lower addresses in the specified stack. If `stack_addr` is NULL the stack is allocated from the heap. If `stack_size` is not zero the stack will be of the specified size. Otherwise a default stack size is used. Zeroed thread-local storage is also allocated to the thread. `thread_create()` returns the ID of the new thread. The thread IDs have meaning only within a process. The initial thread priority and signal mask is set to the same values as its creator. When the new thread is started, it begins execution by a procedure call to `func(arg)`. If `func` returns, the thread exits (calls `thread_exit()`). The `flags` argument provides the following (or'able) options:

THREAD_STOP

The thread is to be immediately suspended after it is created. The thread will not run until another thread executes `thread_continue()` to start it. If `THREAD_STOP` is not specified, the thread is immediately runnable.

THREAD_NEW_LWP

A new LWP is created along with the thread. The new LWP is added to the pool of LWPs used to execute threads.

THREAD_BIND_LWP

A new LWP is created and the new thread is permanently bound to it.

THREAD_WAIT

Specifies that another thread will eventually wait for this thread to exit. This also means that the thread ID of a thread created with `THREAD_WAIT` will not be reused until the waiting thread returns. If the thread is not created with `THREAD_WAIT`, the thread ID may be reused at any time after the thread exits.

Thread concurrency control

`thread_setconcurrency()` sets the degree of real concurrency (i.e. the number of LWPs) that unbound threads in the application require to `n`. The number of LWPs permanently bound to threads is not included in `n`. If `n` is zero (the default), the library automatically creates as many LWPs for use in scheduling unbound threads as required to avoid deadlock. This number can be incremented by creating a thread with the `THREAD_NEW_LWP` flag. If `n` is less than the current maximum, LWPs are removed from the pool.

`thread_setconcurrency()` guarantees only that this degree of concurrency is available to application threads. The actual number of LWPs employed by the library at any one time may vary.

The number of LWPs automatically created by the library ($n = 0$) is sufficient to avoid deadlock, but it may not be enough to avoid poor performance; the library may create too few or too many LWPs. The programmer may tune the number of LWPs by

creating threads with the `THREAD_NEW_LWP` flag or using `thread_setconcurrency()` as required by the application.

Thread termination

`thread_exit()` terminates the current thread and deallocates thread resources allocated by the threads package.

<pre> thread_id_t thread_create(char *stack_addr, unsigned int stack_size, void (*func)(), void *arg, int flags); int thread_setconcurrency(int n); void thread_exit(); thread_id_t thread_wait(thread_id_t thread_id); thread_id_t thread_get_id(); int thread_sigsetmask(int how, sigset_t *set, sigset_t *oset); int thread_kill(thread_id_t thread_id, int sig); int thread_stop(thread_id_t thread_id); int thread_continue(thread_id_t thread_id); int thread_priority(thread_id_t thread_id, int priority); void mutex_init(mutex_t *mp, int type, void *arg); void mutex_enter(mutex_t *mp); void mutex_exit(mutex_t *mp); int mutex_tryenter(mutex_t *mp); </pre>	<pre> void cv_init(condvar_t *cvp, int type, void *arg); void cv_wait(condvar_t *cvp, mutex_t *mutexp); void cv_signal(condvar_t *cvp); void cv_broadcast(condvar_t *cvp); void sema_init(sema_t *sp, unsigned int count, int type, void *arg); void sema_p(sema_t *sp); void sema_v(sema_t *sp); int sema_tryv(sema_t *sp); void rw_init(rwlock_t *rwlp, int type, void *arg); void rw_enter(rwlock_t *rwlp, rw_type_t type); void rw_exit(rwlock_t *rwlp); int rw_tryenter(rwlock_t *rwlp, rw_type_t type); void rw_downgrade(rwlock_t *rwlp); int rw_tryupgrade(rwlock_t *rwlp); </pre>
---	--

Figure 4: Thread interface functions

Waiting for threads

`thread_wait()` blocks until the specified thread exits. It is an error to wait for a thread that was created without the `THREAD_WAIT` attribute, to wait for the current thread, or to have multiple `thread_wait()`s on the same thread. If `thread_id` is `NULL`, then any thread marked `THREAD_WAIT` that exits causes `thread_wait()` to return. If a stack was supplied by the programmer when the thread was created, it may be reclaimed when `thread_wait()` returns successfully. `thread_wait()` returns the ID of the thread that exited if the wait is successful. After `thread_wait()` returns successfully, the returned `thread_id` is unusable in any subsequent thread operation.

An alternate interface for this function is `waitid()` with `id_type` equal to one of the following:

`P_THREAD`

`waitid()` waits for the thread specified by `id`.

`P_THREAD_ALL`

`waitid()` waits for any thread marked `THREAD_WAIT`.

The exit status of a thread is always zero.

Thread identification

`thread_get_id()` returns the thread ID of the caller.

Thread signal mask

`thread_sigsetmask()` or `sigproc-mask()` sets the thread's signal mask.

Thread signaling

`thread_kill()` causes the specified signal to be sent to the specified thread. An alternate interface for this function is `sigsend()` with `id_type` equal to one of the following:

`P_THREAD`

`sig` is sent to the thread within the process specified by `id`.

`P_THREAD_ALL`

`sig` is sent to all the threads within the process.

Thread execution control

`thread_stop()` prevents the specified thread from running. If `thread_id` is `NULL` then the current thread is immediately stopped. `thread_continue()` initially starts a thread or restarts a thread after `thread_stop()`. The effect of `thread_continue()` may be delayed, but `thread_stop()` does not return until the specified thread is stopped.

Thread priority control

`thread_priority()` sets the priority of the specified thread. If `thread_id` is `NULL` the current thread is used. The priority must be greater than or equal to zero. Increasing the specified priority gives increasing scheduling priority. The old priority is returned. If the specified thread is not running then it may or may not execute immediately even though its new priority is greater than a currently executing thread.

Thread synchronization

The thread synchronization facilities are designed to synchronize threads both within a process and between processes. When a synchronization variable is initialized, the programmer must specify whether the synchronization variable is to be shared between processes. The programmer can usually also specify other variants such as extra debugging, spin waiting, etc. The programmer may bitwise-or `THREAD_SYNC_SHARED` into the variant type to specify that the variable is to be shared between processes.

Any synchronization variable that is statically or dynamically allocated as zero may be used immediately without further initialization, and provides the default implementation variant in the default initial state. A dynamic initialization with an implementation variant type of zero also specifies the default implementation variant.

Mutex locks

Mutex locks provide simple mutual exclusion. They are low overhead in both space and time and are therefore suitable for high frequency usage. Mutex locks are strictly bracketing in that it is an error for a thread to release a lock not held by the thread. Mutex locks are used to prevent data inconsistencies in critical sections of code. They may also be used to preserve code that is single threaded.

`mutex_enter()` acquires the lock, potentially blocking if it is already held. `mutex_exit()` releases the lock, potentially unblocking a waiter. `mutex_tryenter()` acquires the lock if it is not already held. `mutex_tryenter()` can be used to avoid deadlock in operations that would normally violate the lock hierarchy.

Condition variables

Condition variables are used to wait until a particular condition is true. Condition variables must be used in conjunction with a mutex lock. This implements a typical monitor.

`cv_wait()` blocks until the condition is signaled. It releases the associated mutex before blocking, and reacquires it before returning. Since the reacquiring of the mutex may be blocked by other threads waiting for the mutex, the condition that caused the wait must be re-tested. Thus, typical

usage is:

```
mutex_enter(&m);
...
while (some_condition) {
    cv_wait(&cv, &m);
}
...
mutex_exit(&m);
```

This allows the condition to be a complicated expression, as it is protected by the mutex. There is no guaranteed order of acquisition if more than one thread blocks on the condition variable.

`cv_signal()` wakes up one of the threads blocked in `cv_wait()`. `cv_broadcast()` wakes up all of the threads blocked in `cv_wait()`. Since `cv_broadcast()` causes all threads blocking on the condition to re-contend for the mutex, it should be used with care. For example, it is appropriate to use `cv_broadcast()` to allow threads to contend for variable amounts of resources when resources are released.

Semaphores

The semaphore synchronization facilities provide classic counting semaphores. They are not as efficient as mutex locks, but they need not be bracketed so that they may be used for asynchronous event notification (e.g. in signal handlers). They also contain state so they may be used asynchronously without acquiring a mutex as required by condition variables.

`sema_p()` decrements the semaphore, potentially blocking the thread. `sema_v()` increments the semaphore, potentially unblocking a waiting thread. `sema_try_p()` decrements the semaphore if blocking is not required.

Multiple readers, single writer locks

Multiple readers, single writer locks allow many threads simultaneous read-only access to an object protected by this lock simultaneously. It allows only one thread to access an object for writing at any one time, and excludes any readers. A good candidate for a multiple readers, single writer lock is an object that is searched more frequently than it is changed. For brevity this type of lock is also known as a readers/writer lock.

`rw_enter()` attempts to acquire a reader or writer lock. `type` may be one of the following:

`RW_READER` Acquire a readers lock.

`RW_WRITER` Acquire a writer lock.

`rw_exit()` releases a readers or writer lock. `rw_tryenter()` acquires a readers or writer lock if doing so would not require blocking. `rw_downgrade()` atomically converts a writer lock into a reader lock. Any waiting writers remain waiting. If there are no waiting writers it wakes up any pending readers. `rw_tryupgrade()` attempts to atomically convert a reader lock into a writer

lock. If there is another `rw_tryupgrade()` in progress or there are any writers waiting, it returns a failure indication.

Lightweight process state

A lightweight process consists of a data structure in the kernel used for processor scheduling, page fault handling, and kernel call execution. It also contains state that is private to the LWP and an association with a process (address space). The following programmer-visible state is maintained by the kernel and is unique to each LWP within a process:

- LWP ID
- Register state (including PC and stack pointer)
- Signal mask
- Alternate signal stack and masks for alternate stack disable and onstack
- User and user+system virtual time alarms
- User time and system CPU usage
- Profiling state
- Scheduling class and priority

All other process state is shared by the LWPs within the process.

Note that even though the CPU usage, virtual time alarms, and alternate signal stack are available to each LWP, this state is not kept for each thread that is multiplexed on LWPs. Threads that require this state must be bound to an LWP. Whether the LWP state includes a separate stack area known to the kernel or not is implementation dependent. Of course, the lightweight process runs with a stack.

Signals

A new signal, `SIGWAITING`, is sent to the process when all its LWPs are waiting for some indefinite, external event (e.g. in `poll()`). The default handling for `SIGWAITING` is to ignore it. The threads package can use the receipt of `SIGWAITING` to cause extra LWPs to be created as required to avoid deadlock. This is similar in functionality to the architecture described in [Anderson 1990].

While `SIGWAITING` is sent for "indefinite" waits, supposedly short term blocking for things like page faults or file system I/O may take a long time relative to the speed of the CPUs. It may be desirable to define an alternate signal that is sent in these cases.

Time, interval timers, and profiling

There is only one real-time interval timer per process, so it delivers one signal to an address space when it reaches the specified time interval. Library routines may implement multiple per-thread timers using the per-address space timer when that

functionality is required. Each LWP has two private interval timers; one decrements in LWP user time and the other decrements in both LWP user time and when the system is running on behalf of the LWP. When these interval timers expire either SIGVTALRM or SIGPROF, as appropriate, is sent to the LWP that owns the interval timer.

Profiling is enabled for each LWP individually. Each LWP can set up a separate profiling buffer, but it may also share one if accumulated information is desired. Profiling information is updated at each clock tick in LWP user time. The state of profiling is inherited from the creating LWP.

Resource usage

The resource limits set limits on the resource usage of the entire process (i.e. the sum of the resource usage of all the LWPs in the process). When a soft resource limit has been exceeded, the LWP that exceeded the limit is sent the appropriate signal. The sum of the resource usage (including CPU usage) for all LWPs in the process is available via `getrusage()`.

Process creation and destruction

The `fork()` system call attempts to duplicate the existing UNIX semantics. It duplicates the address space and creates the same LWPs in the same states as in the original. This duplicates the threads in the original process. Calling `fork()` may cause interruptible system calls to return EINTR when the calls are made by any LWP (thread) other than the one calling `fork()`.

A new system call, `fork1()`, causes the current thread/LWP to fork, but the other threads and LWPs that existed in the original process are not duplicated in the new process. `fork1()` is defined as follows:

```
int fork1();
```

The return values are similar to `fork()`.

Both the `exit()` and `exec()` system calls work as usual, except that they destroy all the LWPs in the address space. Both calls block until all the LWPs (and therefore all active threads) are destroyed. When `exec()` rebuilds the process, it creates a single LWP. The process startup code then builds the initial thread.

Why have both `fork()` and `fork1()`?

UNIX `fork()` seems to have two generic uses; to duplicate the entire process (the BSD dump program uses this technique), or to create a new process in order to set up for `exec()`. For the latter purpose, `fork1()` is much more efficient because there is no need to duplicate all the LWPs. There are, however, dangers to using `fork1()`. First,

since threads are maintained by the threads library as data structures, the threads library must take care that after `fork1()` only the issuing thread remains in the new address space, which is a duplicate of the old one. Secondly, the programmer must be careful to call only functions that do not require locks held by threads that no longer exist in the new process. This can be difficult to determine as libraries can create hidden threads. Lastly, locks that are allocated in memory that is sharable (i.e. `mmap()`'ed with the `MAP_SHARED` flag) can be held by a thread in both processes, unless care is taken to avoid this. The latter problem can also arise with `fork()`.

Having `fork()` completely duplicate the process is the semantic that is most similar to the single-threaded `fork()`. It allows both generic uses and there are fewer pitfalls for the programmer. Having `fork1()`, which forks only one thread, permits optimized `fork()` and immediate `exec()` (e.g. `system()`).

Scheduling

LWPs (and bound threads) can change their scheduling class and class priority via the `priocntl()` system call. A new scheduling class for "gang" scheduling is available for implementations of fine grain parallelism. The LWP may also ask to be bound to a CPU, depending on the scheduling class.

Debugging

The `/proc` file system has been extended to reflect the changes to the process model required by the addition of multi-threading at the process level. Of necessity, a kernel process model interface can provide access only to kernel-supported threads of control, namely LWPs. Debugger control of library threads is accomplished by cooperation between the debugger and the threads library, with the aid of the `/proc` file system to control the kernel-supported LWPs.

The details of the `proc` file system and some of the enhancements for multi-threading support can be found in [Faulkner 1991].

Performance

All the performance numbers in this section were obtained on a SPARCstation 1+ (Sun 4/65), which is a 25Mhz SPARC platform. The measurements were made using the built-in microsecond resolution real-time timer. The numbers reflect an untuned prototype system.

Thread creation time

The first measurement is for thread creation time. It measures the time consumed to create a thread using a default stack that is cached by the

threads package. The measured time only includes the actual creation time, it does not include the time for the initial context switch to the thread. The results are shown in Figure 5. The ratio column gives the ratio of the creation time in that row to the creation time in the previous row.

	Time (usec)	ratio
Unbound thread create	56	-
Bound thread create	2327	42

Figure 5: Thread creation time

Measurements were taken for creating both bound and unbound threads. Bound thread creation involves calling the kernel to also create an LWP to run it. Unbound thread creation is done without kernel involvement.

Thread synchronization time

The second measurement is for thread synchronization time. It measures the time it takes for two threads to synchronize with each other using two synchronization variables, as shown below:

```
sema_t s1, s2;
thread1()
{
    ...
    start_timer();
    sema_v(&s1);
    sema_p(&s2);
    t = end_timer();
    ...
}
thread2()
{
    ...
    sema_p(&s2);
    sema_v(&s1);
    ...
}
```

The numbers presented in Figure 6 are the results of the above measurement divided by two, since there are actually two synchronizations involved. The ratio column gives the ratio of the synchronization time in that row to the synchronization time in the previous row.

	Time (usec)	ratio
Setjmp/longjmp	59	-
Unbound thread sync	158	2.7
Bound thread sync	348	2.2
Cross process thread sync	301	.86

Figure 6: Thread synchronization time

The first measurement is a simple routine that does a `setjmp()` and `longjmp()` to itself. It is presented as a baseline for thread switching time.

The next two measurements are for unbound and bound threads synchronizing within a process. The last measurement is for threads in two different processes synchronizing through a file in shared memory.

Comparison with other Thread Models

This section addresses the similarities and differences between the SunOS multi-thread (MT) architecture and other commercially available multi-thread interfaces. Instead of comparing procedural interfaces, the discussions concentrate on comparing and contrasting architectural issues. The comparisons underscore what we believe are the key differences rather than being comprehensive.

Mach Release 2.5 C Threads

Mach Release 2.5 C Threads [Cooper 1990], [Tevanian 1987] exemplifies a thread interface that provides the programmer with the means to express concurrency, independent of the underlying system support. While this is a desirable trait, Mach 2.5 C Threads does not acknowledge the existence of a second layer of abstraction (i.e. LWPs) and therefore does not allow the programmer to control the degree of kernel resources it uses. In many useful applications the programmer must know and manipulate the degree of actual kernel resources required. For example, a window system programmer must know that extremely lightweight threads are available, since a window system may use thousands. A micro-tasking Fortran run-time library relies on kernel-supported threads that are scheduled on processors as a group. Database programmers may require a mixture of the two situations. In addition, there may be aspects to kernel-supported threads that are too "heavyweight" to export to lightweight threads (e.g. virtual time) and are required by some applications.

In Mach 2.5, C Threads libraries have been constructed that map threads directly to kernel-supported threads or multiplex threads on kernel-supported threads, but one application cannot have both types at the same time. In addition, there can be no direct access to "heavyweight" features of kernel-supported threads since that would allow only a one-to-one mapping between threads and kernel-supported threads.

Newer versions of Mach [Golub 1990] have corrected some of these deficiencies by extending the C Threads interface to provide a two-level model similar to ours. In the new library, C Threads are multiplexed on Mach kernel threads. In addition, new C Threads interfaces allow C Threads to bind to Mach kernel threads.

The main difference between the C Threads synchronization primitives and the SunOS MT architecture primitives is the scope of operation. C

Threads does not explicitly support the use synchronization variables allocated in mmap'ed memory even though Mach virtual memory supports the sharing of memory between tasks. The SunOS MT architecture supports this and also allows the placement of synchronization variables in files to control access to the file data, and having the lifetime of such synchronization variables be greater than that of the creating process.

C Threads supports per-process signal state. There is no per-thread signal mask. There is no way for a thread to control when it can handle a signal except by preventing all the threads in a process from handling it. When a particular thread is in a critical section of code with respect to the signal handler, it must block the interrupt for all threads. This can cause severe performance problems in heavily asynchronous applications. The alternate solution for C Threads is Mach IPC. Mach IPC, however, does not allow asynchronous interruption of a computation. For example, an application that creates a thread to perform some long computation may wish to terminate the computation regardless of results. There is no way to interrupt the computation unless it is coded to occasionally poll for IPC. This forces the programmer to change the computation code so that polling is done frequently enough to respond to a termination request but not so frequently as to slow down the computation.

Chorus

Chorus [Armand 1990] intentionally avoided user-level threads because of a perceived impact on real-time requirements. For example, the two levels of scheduling interfere with the requirement that the highest priority runnable thread is always allowed to run. SunOS meets this requirement by allowing a thread to bind to an LWP and thus achieve a system-wide scheduling priority. In addition, the bound thread can ask that the underlying LWP be made a member of a real-time scheduling class, which provides more exact scheduling control.

Chorus threads each have a signal mask and a vector of signal handlers. The effect of receipt of an asynchronously generated signal and combinations of catching, SIG_DFL, and SIG_IGN are computed. If one or more threads are catching the signal, it is delivered to all catching threads (broadcast delivery). Otherwise, if any thread has set the handler to SIG_IGN, the signal is discarded. Otherwise the default action is taken on the process. The main deficiency in this model is that broadcast delivery can cause "synchronization storms" when the handling threads try to synchronize. It also causes much extra work for the kernel. Lastly, broadcast makes the number of signals delivered to a process uncountable in a non-queuing signal implementation. For example, if several threads are waiting for a keyboard interrupt, and two are sent, some threads

will receive two signals while others will receive one.

The per-thread signal handlers add some code modularity, at the cost of complexity in the handling of SIG_DFL and SIG_IGN as noted above. The modularity added is relatively minor because asynchronous signals are mostly controlled by the application, not the library. In addition, serial handling of the same signal within a thread is still a problem, just as it is in single-threaded UNIX.

University of Washington.

The variant of the Topaz [McJones 1989] operating system by the University of Washington [Anderson 1990] implements a portable threads interface with lightweight user-level threads that use kernel resources only as required. In most cases threads can synchronize without kernel involvement, while at the same time, I/O, page faults, and other blocking operations do not stop the entire process. This approach has the same advantages as our threads multiplexed on LWPs. However, programmer control over the use of kernel resources is not supported.

The main underlying difference between the University of Washington work and the SunOS MT architecture is that the University of Washington work uses lightweight "scheduler activations" that do upcalls into user space to give schedulable execution contexts to the threads package. An upcall by a new scheduler activation informs the threads package whenever a scheduler activation currently in use by the process blocks in the kernel. This gives the threads package the opportunity to schedule another runnable thread. This is similar to the function of the new SIGWAITING signal in our architecture. This signal also gives the threads library the opportunity to schedule a runnable thread by first creating a new LWP. The main difference is that the current definition of SIGWAITING is much more coarse than the way scheduler activations are used. The former is sent only when the LWP blocks in an indefinite wait. The latter is sent whenever the thread blocks in the kernel for any event. In the future, we plan to experiment with sending signals on "faster" events.

The University of Washington approach gives much finer-grained control over scheduling threads on processors, though it is not clear that this is an absolute requirement. In general, the SunOS MT architecture satisfies most of the requirements that motivated the University of Washington group. The critical observation made by both efforts was that the kernel need not be invoked for every thread operation.

POSIX P1003.4a

Comparison with POSIX P1003.4a Pthreads [POSIX 1990] is somewhat difficult at this time, as it is a moving target. Currently (pre Draft 10) it seems that the signal model is a direct superset of the SunOS model. In addition, there seems to be support for the two-level threads model in the scheduling interfaces. However, the interaction between synchronization variables and mapped files (P1003.4) is missing.

Sun LWP library.

The Sun LWP library [Kepecs 1985] supplied in SunOS 4.0 is a classic user-level-only threads package. It contained no explicit kernel support. Threads (called LWPs) synchronized with each other without kernel involvement. If an LWP called a blocking system call or took a page fault, the entire application blocked. This could be mitigated somewhat by using a non-blocking I/O library instead of the standard UNIX I/O interfaces. The non-blocking I/O library uses kernel-supported asynchronous I/O facilities to mimic standard I/O interfaces and allows the package to switch LWPs when one blocked on an indefinite I/O. The application still blocked when a page fault was taken.

The SunOS multi-thread architecture completely supersedes this interface in functionality.

Summary

The SunOS multi-threading architecture provides the following advantages:

- The two level (threads and LWP) model allows the programmer to decouple logical program parallelism from the relatively expensive kernel-supported parallelism. Programmers can rely on the availability of extremely lightweight threads.
- The architecture allows the programmer to control the degree of real concurrency the application requires or allows the threads package to automatically decide this.
- The architecture has a uniform synchronization model between threads both inside and outside a process.
- The programmer can control the mapping of threads onto LWPs to achieve particular performance or functionality without leaving the threads model.
- The programmer can control the allocation of stacks and thread-local storage. This allows coexistence with different memory allocation models (e.g. garbage collection).
- A minimalist translation of the UNIX environment to threads allows higher-level interfaces such as POSIX Pthreads to be implemented on top of SunOS threads.

References

- [Anderson 1990] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", Department of Computer Science and Engineering, University of Washington, Technical Report 90-04-02, April 1990.
- [Armand 1990] F. Armand, F. Herrmann, J. Lipkis, M. Rozier, "Multi-threaded Processes in Chorus/MIX", Proc. EUUG Spring 1990 Conference, Munich, Germany, April 1990.
- [Cooper 1990] E.C. Cooper, R.P. Draves, "C Threads", Department of Computer Science, Carnegie Mellon University, September 1990.
- [Faulkner 1991] R. Faulkner, R. Gomes, "The Process File System and Process Model in UNIX System V", Proc. 1991 USENIX Winter Conference.
- [Golub 1990] D. Golub, R. Dean, A. Florin, R. Rashid, "UNIX as an Application Program", Proc. 1990 USENIX Summer Conference, pp 87-95.
- [Kepecs 1985] J. Kepecs, "Lightweight Processes for UNIX Implementation and Applications", Proc. 1985 USENIX Summer Conference, pp 299-308.
- [McJones 1989] P.R. McJones and G.F. Swart, "Evolving the UNIX System Interface to Support Multithreaded Programs", Proc. 1989 USENIX Winter Conference, pp 393-404.
- [POSIX 1990] POSIX P1003.4a, "Threads Extension for Portable Operating Systems", IEEE.
- [Tevanian 1987] A. Tevanian, R.F. Rashid, D.B. Golub, D.L. Black, E. Cooper, and M.W. Young, "Mach Threads and the UNIX Kernel: The Battle for Control", Proc. 1987 USENIX Summer Conference, pp 185-198.

Michael L. Powell has been a Distinguished Engineer at Sun Microsystems for three years, and is currently a member of Sun Laboratories. His work includes distributed, multi-threaded, object-oriented software, operating systems, architecture, and compilers. Before joining Sun, he taught at UC Berkeley and worked at DEC Western Research Lab. He has a Ph.D. in Computer Science from Stanford University.

Steve Kleiman is currently a Distinguished Engineer in the Operating Systems Technology Department of Sun Microsystems. He received an M.S. in Electrical Engineering from Stanford University in 1978 and a B.S. in Electrical Engineering and Computer Science from M.I.T in 1977. He has been involved with the design and development of UNIX and workstation architecture since 1977; first at Bell Telephone Laboratories and then at Sun. He was one of the developers of NFS, Vnodes, and of the original port

of SunOS to SPARC.

Steve Barton is currently a Member of Technical Staff at Sun Microsystems. He received a B.A. in Computer and Information Science from the University of California, Santa Cruz in 1982. Since then he has also been at Zilog, Inc., Parallel Computers, CounterPoint Computers, and TeleStream Corp.

Devang Shah is currently a Member of Technical Staff at Sun Microsystems. He joined Sun after completing his M.A. in Computer Science from the University of Texas at Austin. As part of his Master's Thesis he developed UTIX (an extension of SunOS 3.2) which provides a kernel supported light weight process programming model. Prior to UT-Austin, he worked at Tata Consultancy Services, Bombay, India. He received his B. Tech.(Hons.) in Electronics Engineering from the Institute of Technology, B.H.U, India in 1985.

Daniel A. Stein is currently a member of technical staff at Sun Microsystems where he is one of the developers of the multi-threaded SunOS architecture. He received a B.S. from the University of Wisconsin in 1981.

Mary Weeks has been a member of technical staff at Sun Microsystems since 1986. Prior to her participation in the SunOS MT project, she worked on the asynchronous i/o interface and the shared libraries. She received her B.A. in computer science from University of California at Berkeley in 1984.

Bringing the C Libraries With Us into a Multi-Threaded Future

Michael B. Jones – Carnegie Mellon University

ABSTRACT

An enormous amount of UNIX (and UNIX-like) code has been written (by a likewise enormous amount of programmers) that uses the standard C libraries. Use is made throughout much of this code of the knowledge that traditional UNIX programs have exactly one thread of control. However, increasing numbers of UNIX-like systems are beginning to provide support for programs with multiple threads of control. To the extent possible, it is highly desirable to preserve the existing C library interfaces for multi-threaded programs; this will aid both in code and programmer portability between traditional UNIX environments and new multi-threaded ones.

A number of issues must be confronted in order to produce versions of the C libraries which can be used in multi-threaded programming environments. Among these are: functions with non-reentrant interfaces, functions which maintain state between invocations, use of macros in the library interfaces, interactions with signals, compatibility with single-threaded library data structures, performance issues, and of course, errno. Despite these and other problems, experience has shown that reasonable solutions are available. This paper presents both a detailed explanation of the problems inherent in producing multi-thread-safe C libraries and the different solutions which are available. Finally, the solutions to these problems adopted by a number of research and industry groups are presented.

Introduction

Existing C Libraries

An enormous amount of UNIX (and UNIX-like) code has been written that uses the standard C libraries. Such libraries include libc, libm, libtermcap, libcurse, etc. Likewise, there is a wide base of programmer experience with these existing interfaces.

Use is made throughout much of this code of the knowledge that traditional UNIX programs have exactly one thread of control. However, increasing numbers of UNIX-like systems are beginning to provide support for programs with multiple threads of control.

Multi-Threaded UNIX-like Systems Appearing

Systems by both research and industry groups such as Apollo, Mach, Encore, Convex, Chorus, OSF/1, future System V releases, and future SunOS releases are all supporting multi-threaded address spaces. The P1003.4a IEEE POSIX standards group has begun balloting a proposed threads interface standard.

Several advantages of multiple threads over the traditional single-threaded programming model have driven the industry towards supporting multi-threaded programs. Chief among them are:

- Threads provide a natural paradigm for expressing inherent program parallelism; they provide a synchronous alternative to

asynchronous interruption or polling.

- Threads allow for the efficient utilization of multiple processors, when available.

Clear Need for C Libraries Usable in Multi-Threaded Programs

To the extent possible, it is highly desirable to preserve the existing C library interfaces for multi-threaded programs. This will maximize code portability and reusability between traditional UNIX environments and new multi-threaded ones. Possibly even more importantly, this will also maximize programmer portability between the old and new environments by allowing programmers to continue to use already familiar programming facilities.

General Requirements on Multi-Threaded Libraries

Operating in a multi-threaded programming environment imposes additional requirements upon libraries which are not present for single-threaded environments. The main new requirement is that concurrent accesses to data structures shared between threads must be synchronized in such a way that the data structure invariants are maintained. Also, some processor and bus cache architectures require explicit cache synchronization operations to be performed in order to provide a consistent view of shared data structures.

A wide range of different synchronization mechanisms may be used for protecting shared data structure accesses. Among them are mutual exclusion locks (mutexes) and condition variables, spin locks, binary semaphores, counted semaphores, reader-writer locks, and Ada-like rendezvous mechanisms. While the different mechanisms provide somewhat different semantics, each can actually be implemented in terms of the others, giving them equivalent semantic power. However, efficiency considerations make some choices better than others.

Probably the most important criteria for choosing among synchronization mechanisms is that uncontested accesses to shared data should introduce almost no additional overhead. Any of mutexes and condition variables, spin locks, semaphores, and reader-writer locks can meet this requirement; each might be appropriate in different application contexts. Each rendezvous, however, typically requires at least one thread context switch, introducing extra overhead even in the best case. The mutexes and condition variables shall be used in illustrations for the remainder of this paper, recognizing that other synchronization primitives could also have been used.

Locking Approaches

Two general approaches can be taken towards performing synchronization (which henceforward shall also be called simply "locking") for functions which require restrictions upon concurrent execution for correct operation.

- **Internal Locking:** Concurrent execution restrictions are enforced by placing appropriate lock calls within the function itself. No additional preconditions are placed upon callers.
- **External Locking:** Concurrent execution restrictions are enforced by requiring that callers ensure improper concurrent calls are not made. Typically, this means that callers must perform explicit synchronization operations around calls.

A simple example serves to illustrate the difference between the two approaches. The function `put()`:

```
void put(char val)
{
    *put_ptr++ = val;
}
```

which would be called as:

```
put(val1);
```

cannot be safely concurrently executed since the pointer fetch, value store, pointer increment, and pointer store will not be atomically executed.

With internal locking, the function would be modified to execute correctly in the presence of concurrent calls as follows:

```
void put(char val)
{
    mutex_lock(&put_mutex);
    *put_ptr++ = val;
    mutex_unlock(&put_mutex);
}
```

Since the function itself is ensuring mutual exclusion, the critical section is never executed concurrently.

With external locking, the function would be unmodified, but callers would typically have to place locks around calls, as in:

```
mutex_lock(&put_mutex);
put(val1);
mutex_unlock(&put_mutex);
```

Since the callers are ensuring mutual exclusion, the critical section is never executed concurrently.

Discussion of Internal Locking

Some of the advantages of internal locking are:

- **Simplicity of Use:** Such functions' invariants are maintained by the functions' implementations, rather than by their callers, placing no extra semantic burden the callers.
- **Modularity of Use:** Calls to functions utilizing internally locking can be written independently since independent callers need not coordinate with one other to maintain internal function invariants.
- **Existing Function Interfaces:** Existing function interfaces can often be used without modification. Since internal locking makes synchronization issues the responsibility of the implementation, callers need not be changed to address them.
- **Minimal Code Size:** Locking code for each function will typically exist only within the function, and not be potentially replicated at each call site.
- **Program Robustness:** A function's data structures cannot be corrupted by concurrent calls (provided, of course, that the internal locking needed by the function was implemented correctly). The burden of maintaining the function's invariants is on the function's implementer, rather than the function's callers.

The main disadvantage of internal locking is:

- **Extra Locking:** Internal locking potentially performs more locking than is actually needed. By definition, locking is being performed at a granularity necessary to independently protect each such function's invariants across every call. In some circumstances, however, significantly less locking could be performed by performing explicit locking around a group of otherwise unprotected calls, rather than implicitly performing locking internal to each call.

Discussion of External Locking

Some of the advantages of external locking are:

- **Flexible Locking Granularity:** Locking granularity decisions are left up to the application. Thus, less total locking may potentially be performed than that which would have been by internal locking. For instance, applications might place several calls to functions requiring external locking within a critical section protected by a single lock.
- **Existing Function Implementations:** Existing function implementations can be used without modification. Since external locking makes synchronization issues the responsibility of the callers, implementations need not be changed to address them. Of course, to the extent that one library function is a client of another which requires locking, the client function's implementation still must be changed. Thus in practice, many implementations will still have to be changed even if external locking is used.

Some of the disadvantages of external locking are:

- **Increased Code Complexity:** Clients of functions requiring external locking are required to be aware of and maintain concurrency invariants via explicit programming discipline. Explicit locking mechanisms must be introduced in order to maintain these invariants. Furthermore, all modules using such functions must use the same locking mechanisms for the locking to be effective. The result of this is that the external locking mechanism to be used for each such function must also be specified with the function in order for such functions to be usable in a modular fashion.
- **Increased Code Size:** There will typically be a set of locking operations needed around each call to a function needing locking. This makes the amount of locking code needed proportional to the number of calls made to such functions (rather than to the number of such functions called).
- **Undetected Race Conditions:** External locking potentially increases the possibility of undetected race conditions in programs which appear to (sometimes) work. If a programmer forgets to lock around calls to functions requiring external locking, the program will still sometimes work since concurrent calls may not have actually been executed. Sometimes, however, concurrent calls will have been executed, violating the functions' preconditions, resulting in arbitrarily bad results (such as corrupted data structures or illegal memory references). Functions requiring external locking do not fail gracefully when called in subtly incorrect ways, often

making debugging such code a nightmare.

Approaches Not Mutually Exclusive

While it might at first appear that internal and external locking are mutually exclusive choices, this is not actually the case. There are contexts where both might appropriately be used together, gaining the advantages (and disadvantages) of both. For instance, both approaches are used together in many stdio implementations.

Cache Synchronization Techniques

Some processor and bus cache architectures require explicit cache synchronization operations to be performed in order to provide a consistent view of shared data structures. Two different cache synchronization approaches can be taken:

1. **Explicit Flushing:** This approach requires applications to perform explicit cache synchronization operations whenever shared memory needs to be made consistent. The advantage of this approach is that the application may be able to perform the least possible number of such synchronization operations. The disadvantages are that such operations are inherently non-portable and error-prone.
2. **Flushing by Locking Primitives:** This approach requires the locking primitives to perform any cache synchronization operations necessary in to provide a consistent view of the data structures protected by the locks. This is a logical approach since acquiring a lock expresses intent to modify a shared data structure, and releasing it expresses that the modification has been completed. These are the two points where cache synchronization operations would likely be required anyway. While in theory extra synchronization might be performed, in practice it would have been needed in any event. Also, hiding the cache synchronization operations removes a source of non-portability.

Problems with Existing C Libraries

Many of the functions provided by the standard C libraries are already reentrant and thread-safe. Nonetheless, a number of problems must be overcome to make versions of the standard C libraries which are usable in multi-threaded programming environments. The following sections outline some of the problems and solutions.

Functions with Non-Reentrant Interfaces

A number of functions (e.g., `localtime()`, `open_dir()`, `getpwent()`, etc.) have interfaces which are non-reentrant because they return results in statically allocated buffers or structures. Such functions cannot be safely used in multi-threaded programs in an

unrestricted fashion since a call by one thread may overwrite the results of a call by another thread before they are used. A number of different solutions are possible:

1. **External Locking:** This approach requires clients to hold locks when calling such functions and using the results. For instance, a mutex could be introduced for each such static result area which clients would be required to hold. A call to `localtime()` might then look like:

```
struct tm *tp;
mutex_lock(&localtime_mutex);
tp = localtime(&clock);
/* Use results pointed to by tp */
mutex_unlock(&localtime_mutex);
/* Results at *tp potentially
   corrupted */
```

2. **Thread-Specific Data:** This approach has such functions return pointers to thread-specific data areas instead of statically allocated data areas. This would require such functions to be modified to allocate thread-specific storage. This approach has the advantage that calls remain the same and no extra client locking is needed. One disadvantage is the potential performance penalty of thread-specific data manipulation. Another disadvantage is the extra per-thread storage space required for each such result area; unless garbage collection is being used, thread-specific storage will not be freed until at least thread exit time, and possibly not until the program exits.
3. **Alternate Reentrant Versions:** This approach provides alternate reentrant versions of such functions. These versions would be passed pointers to result areas allocated by the caller. For instance, a reentrant version of `localtime()` might look like:

```
int localtime_r(struct tm *result,
               time_t *clock);
```

One unfortunate point about some C implementations should probably be noted here. Some C compilers (for instance, those which are derived from *pcc*) return structure values from functions in statically allocated storage. Thus, functions which may appear to be reentrant are actually not when compiled with some compilers. Fortunately, such functions are typically not used by the traditional C libraries.

Functions which Maintain State Between Invocations

A number of functions (e.g., `rand()`, `malloc()`, `getpwent()`, the stdio functions, etc.) maintain state between invocations. One problem which must be resolved in a multi-threaded environment is whether

such state should be maintained on a per-thread basis, a per-process basis, or whether allocation of the state should be placed under application control. A number of different solutions are possible:

1. **External Locking:** This approach requires clients to hold locks when calling such functions. The locks guard manipulations of the persistent state. This presumes that the persistent state should be maintained on a per-process basis. In the case of `rand()`, for instance, this means that a single random number stream would be shared by all threads.
2. **Internal Locking:** This approach performs any locking necessary for manipulation of persistent state within the function manipulating it. This approach also presumes that the persistent state should be maintained on a per-process basis.
3. **Thread-Specific Data:** This approach maintains the persistent state in thread-specific storage areas. In the case of `rand()`, for instance, this means that a different random number stream would be used by each thread. Issues of thread-specific storage allocation, initialization, and deallocation must be addressed with this approach.
4. **Alternate Reentrant Versions:** This approach places the allocation of persistent state under program control. This is the most general solution for such functions. This can be done by providing alternate reentrant versions of such functions with explicit parameters for all state. For instance, new versions of `rand()` and `srand()` might look like:

```
#include <stdlib.h>
/* Defines rand_state_t */
int rand_r(rand_state_t *s);
int srand_r(unsigned int seed,
            rand_state_t *s);
```

This is sufficiently general for implementing per-thread, per-process, and other sharing schemes, depending upon the allocation and use of the persistent state parameter.

The stdio Functions

Locking for the stdio Functions

The I/O buffers manipulated by the stdio functions are a special case of state which persists across function invocations. In theory, any of the approaches outlined above could be taken for these functions. In practice, thread-specific buffer allocation can probably be immediately ruled out, as the consequence would be that stdio calls (e.g., `getc()`, `printf()`) from different threads would operate on different I/O streams even though they used the same "FILE *" parameter; `stdin`, `stdout`, and `stderr` would mean different things from different threads! Per-process buffer allocation is almost certainly

sufficient.

Next, note that no new reentrant versions of the stdio functions need be introduced since they can already be made reentrant. The "FILE *" parameters already serve as a handle for identifying each set of persistent state, with *fopen()*, *fclose()*, etc., serving as general purpose allocators and deallocators. Also note that while the I/O buffers are allocated on a per-process basis, more flexible usage schemes are certainly possible, depending upon how the "FILE *" pointers are shared between threads.

This leaves either external locking or internal locking. Internal locking has the advantages that typical client code need not perform any extra locking and that the stdio data structures would not be corrupted by erroneous unlocked calls. External locking has the advantage that applications can explicitly gain and relinquish control of I/O streams, preventing unwanted interleaved reads or writes. A combination of these approaches with the advantages of both has actually been used by several implementations.

The combination locking approach taken by some implementations for the stdio functions internally locks the stdio streams while manipulating them, protecting the internal data structure integrity and allowing clients to call the stdio functions (e.g., *printf()*) without holding explicit locks. It also allows clients to explicitly lock stdio streams across stdio calls, preventing unwanted interleaved use by multiple threads. To accomplish this, the implementation uses and exports a recursive (or counted) mutex which allows a thread to recursively reacquire and release the same lock without deadlocking, only actually releasing the lock when the lock count goes to zero.

The new stdio lock functions are of the form:

```
void flockfile(FILE *stream);
void funlockfile(FILE *stream);
```

and allow usage of the form:

```
flockfile(f);
fprintf(f, "in");
fprintf(f, "divisible");
funlockfile(f);
```

sending the characters "indivisible" to stream *f* without any intervening characters possibly being inserted.

The putc(), getc(), putchar(), and getchar() Macros

The *putc()*, *getc()*, *putchar()*, and *getchar()* macros present special problems. If internal locking is to be used, then the macro definitions themselves must be changed to add it or call locking versions. Unlike most other changes, this requires recompilation instead of just relinking.

Also, the reason that these functions are implemented as macros rather than subroutines in the first place is performance. While locking overhead may be small relative to the work done by most functions, it will be somewhat larger relative to the tiny amount of work done by these macros. For some programs, this extra overhead may be unacceptable.

Four alternatives are possible:

- 1 **Fast but Unsafe:** Don't introduce locks into *putc()*, etc. All calls would then have to occur inside explicit *flockfile()* and *funlockfile()* scopes so that necessary locking would be done; otherwise the stdio data structures could be corrupted. Existing code would have to have the lock calls added to be converted.
- 2 **Slow but Safe:** Introduce locks into *putc()*, etc. Explicit locking would be unnecessary, but each call would incur lock overhead. Existing code would not have to be changed.
- 3 **Safe allowing Fast:** Introduce locks into *putc()*, etc. as in the "slow but safe" choice, but also provide versions of them without locks as in the "fast but unsafe" choice using different names. For instance, some implementations use the names *putc_unlocked()*, *getc_unlocked()*, etc. for versions without locking, and the normal names for the locking versions. Existing code can be recompiled and will be correct. The unlocked versions can be used inside explicitly locked scopes where performance is critical. An example using the unlocked versions is:

```
flockfile(stdout);
putchar_unlocked('O');
putchar_unlocked('K');
putchar_unlocked('\n');
funlockfile(stdout);
```

From a software engineering point of view, this approach is the clear winner. Existing code recompiles and works; no unexpected race conditions are introduced. Making the macros thread-safe is consistent with the treatment of the other stdio functions. Plus, the fast versions are available for contexts where the extra performance is critical.

- 4 **Fast allowing Safe:** This is similar to the "safe allowing fast" choice, except that the default is the unsafe versions, with the safe versions provided under new names. It has all the drawbacks of "fast but unsafe" and adds very little since the safe versions could already be constructed by using explicit lock calls around the unsafe versions. It is included in this list largely for the sake of completeness.

The *errno* Variable

The *errno* variable presents a problem which is a special case of functions with non-reentrant interfaces; it is a result returned in statically allocated storage from a large number of functions. The same solutions apply, namely:

- 1 **External Locking:** This approach requires an "errno lock" to be held when calling functions using *errno* and using the results. This would present an immense, tedious burden to programmers, serialize nearly all multi-processor system calls, and require changing all code making system calls when converting it to be multi-threaded.
- 2 **Thread-Specific Data:** This approach allocates a thread-specific *errno* value for each thread. This requires modification of the *errno* variable declaration to contain an address calculation using the current thread as an implicit parameter. One such declaration might be:

```
extern int *_errno(void);
#define errno (*_errno())
```

where the *_errno()* function returns the thread-specific *errno* address for the calling thread. Fortunately, ANSI C specified that the *errno* declaration is provided by the header file "<errno.h>" in anticipation of such modifications. Calls to functions returning values in *errno* would not have to be changed; likewise, code fetching the value of *errno* would remain unchanged.

- 3 **Alternate Reentrant Versions:** This approach provides alternate reentrant versions of functions using *errno* to return results. These versions would be passed a pointer to an int for storing the *errno* value. For instance, a reentrant version of *kill()* might look like:

```
int kill_r(int pid, int sig,
           int *error);
```

This would require changing all calls to functions returning values in *errno* when converting them to be multi-threaded.

Functions with Non-Reentrant Implementations

Some existing libraries contain non-reentrant implementations of functions even though they have reentrant interfaces. One common problem area is mathematical function implementations; such functions often store temporary results in statically allocated variables, particularly when written in assembly language. Software floating point implementations are often culprits. Such implementations need to be fixed when creating multi-threaded libraries.

As pointed out earlier, some compilers generate needlessly non-reentrant code as well (e.g., returning structure values in statics). The only solutions

possible in this case are either to fix the compiler or avoid using the broken compiler features entirely.

Signals in the Multi-Threaded Environment

Signals cause a whole range of potential complications for multi-threaded programs. These problems stem from several causes:

- Signals have been heavily overloaded to perform several unrelated functions (e.g., synchronous exception handling, low-bandwidth asynchronous IPC, process control).
- Signal semantics vary significantly between different implementations and even between different signals in the same implementation.
- Signals can potentially cause asynchronous interruption of a program's critical sections.

A number of different approaches to signal handling are possible in multi-threaded programs. Most make a distinction between two types of signals:

- **Synchronously Generated Signals:** These signals are attributable to a specific thread. For example, executing an illegal instruction or touching invalid memory causes a synchronously generated signal.
- **Asynchronously Generated Signals:** These signals are not attributable to a specific thread. For example, signals sent via *kill()* or from the keyboard are asynchronously generated.

Most approaches deliver synchronously generated signals to the thread which generated them. For asynchronously generated signals an enormous space of behaviors is possible.

An outline of the major decisions which can be made concerning signals is:

- Are synchronous facilities provided for handling asynchronously generated signals?
- Are asynchronous signal handlers supported in multi-threaded programs? If so,
- Are (a) handler vectors, (b) signal masks, and (c) pending bits, maintained on a per-thread or per-process basis?
- Are asynchronously generated signals delivered to (a) a single distinguished thread, (b) a single arbitrarily chosen thread, (c) a single "interested" thread, (d) all "interested" threads, (e) all threads, (f) some combination of the above, or (g) none of the above?

To further muddy the waters, note that a SIGALRM sent as a result of an *alarm()* call might logically be considered to be attributable to the thread which called *alarm()*, making it very much like a synchronously generated signal, even though it is delivered asynchronously! And remember, of course, that the traditional implementation of *sleep()* (which should cause the calling thread to sleep) uses SIGALRM (which may or may not be delivered on a

per-thread basis).

Suffice it to say that signals and multi-threaded programs taken together are complicated. Those desiring a more detailed treatment of the subject should consult the POSIX threads (pthreads) draft.

Areas Possibly Requiring New Functionality

Some problems come up in multi-threaded programs which wouldn't otherwise. This section examines two such areas and presents possible solutions.

Thread-Specific Data

Many types of computations require maintaining some amount of state on a per-thread basis. Such data is logically thread-specific, and can be thought of as additional context with which the thread executes. Single-threaded programs have no need for a thread-specific data mechanism since global or static storage can be considered to be "specific" to the single "thread".

Typically programs will require a number of different thread-specific data areas of different sizes and data types. Independently written modules must be able to allocate their own thread-specific data variables if the mechanism is to be usable in a modular fashion.

Several different thread-specific data mechanisms are possible:

- 1 **Keyed on Thread-ID:** The most primitive type of thread-specific data support requires only that each thread have some form of identifier which is available at runtime which can be used as a hash or index value. Hash tables keyed on the thread-id can then be constructed to contain thread-specific data values.
- 2 **Fixed Set of Locations:** Some thread packages provide a small fixed number of thread-specific data locations to the application. These can be used to construct a more general thread-specific data mechanism by using one such location to hold a pointer to larger, possibly extensible data structures.
- 3 **General Key-Value Mechanism:** Some thread packages provide facilities for allocating new thread-specific variable "keys" which are used to refer to thread-specific data values. Logically, each (key, thread-id) pair serves as an index into the set of values. Some mechanisms of this type also provide facilities for calling destructor functions on thread exit to perform any necessary thread-specific data cleanup. Alternatively, garbage collection could also be used to effect cleanup.
- 4 **Compiler and/or Architectural Support:** Some systems provide compiler and/or architectural support for thread-specific data.

Some compilers, for instance, have introduced a "thread" storage class which is analogous to "static" and "auto". Some systems provide for thread-specific virtual memory mappings, allowing some virtual addresses to refer to thread-specific data. While this can be efficiently implemented on some hardware when threads are mapped directly onto processors, it means that each thread requires a full-weight kernel context, precluding lighter-weight coroutine-like thread contexts. Other systems support thread-specific data through use of a dedicated register.

Orderly Cancellation Mechanism

One style of multi-threaded programming allows a thread to cancel the computation being performed by another. The prototypical example application is parallel chess search: One thread has found a "best" move and thus can cancel the searches for moves which other threads are performing. Such facilities are unnecessary for single-threaded programs since there are no other threads to cancel.

One possible kind of cancellation is to simply stop a thread wherever it is and return its stack, etc. to the free pool. Unfortunately, this is inadequate and unsafe; if a thread is holding resources (e.g., locks, file descriptors, dynamic memory, etc.) when canceled then they will never be released. This can cause deadlocks, resource leakage, or both.

At the opposite extreme, threads could be required to poll for cancellation. Unfortunately, this places both undue complexity and performance burdens on code which needs to be cancelable.

With orderly cancellation it should be possible to:

- establish handlers in program scopes which need to perform cleanup upon cancellation; this makes orderly cancellation possible.
- efficiently control the points at which cancellation may occur; this keeps the cleanup complexity manageable.

Not too surprisingly, this begins to look a lot like a general exception handling facility coupled with the ability to raise a cross-thread exception.

Two main approaches can be taken for supporting cancellation:

- 1 **Build from Existing Facilities:** It is possible to construct an orderly cancellation facility given a mechanism to asynchronously change the execution state (i.e., program counter) of another thread. For instance, certain kinds of per-thread signal support are sufficient. Cleanup handlers can be implemented as procedures which are logically pushed on scope entry (for scopes needing cleanup) and popped on scope exit.
- 2 **Compiler Support:** Compiler support for

exception handling and stack unwinding makes the bulk of cancellation trivial. Canceling a thread then becomes simply raising an exception in the thread to be canceled.

Other Multi-Threaded Library Issues

The Compilation Environment

A number of approaches are possible for providing a compilation environment for building multi-threaded programs. These approaches vary depending upon whether or not they continue to also support building traditional single-threaded programs and if so, how they do so. Some approaches are outlined below.

- **Single compilation environment:** This approach provides a single compilation environment for both single-threaded and multi-threaded programs which combines the facilities used by both kinds of programs. Simplicity is a major advantage of this approach. The main drawback is that single-threaded programs will use thread-safe versions of many library functions, incurring some unnecessary lock overhead. Another disadvantage is that multi-threaded programs can mistakenly call functions which are only usable in single-threaded programs.
- **Alternate compilation environment:** This approach provides a separate compilation environment for multi-threaded programs. It provides alternate versions of some include files, libraries, and possibly programs (e.g., "`<stdio.h>`", "`<stdlib.h>`", "`libc.a`", `cc`, etc.). The alternate versions could be selected at build time either by using search paths (with separate search paths for finding include files, libraries, and programs) or via explicit "`-I`" and "`-L`" switches to `cc`.

This approach allows both single and multi-threaded programs to be built with appropriate headers and libraries: single-threaded programs pay no unnecessary lock costs; multi-threaded programs can only call multi-thread-safe library functions. Attempts to use inappropriate functions can be detected either at compile time or at link time. The main disadvantage of this approach is the complexity of establishing the correct search paths. Space is also required for two versions of many libraries.

- **Conditional compilation environment:** This approach selects additional or changed features needed by multi-threaded programs via compile-time switches and separate library names. For instance, some implementations require the symbol "`_REENTRANT`" to be defined when compiling multi-threaded programs, and use the suffix "`_r`" (e.g., "`libc_r.a`") for identifying reentrant versions

of libraries. This allows a single set of include files to be used for both types of programs.

Like the "alternate compilation environment" approach, both single and multi-threaded programs may be built with appropriate declarations and libraries. Inappropriate calls may be detected at compile or link time. Having a single set of include files increases simplicity. Also, a compiler option could be implemented which automatically defined the appropriate compiler symbols (e.g., "`_REENTRANT`") and rewrote library names to select reentrant versions (e.g., "`-lc`" to "`-lc_r`"), making the environment selection process nearly foolproof. The one drawback of this approach is that space is required for two versions of many libraries.

A strong argument can be made against functions which are only usable in single-threaded programs being available when building multi-threaded programs; if the functions can't be safely used they can only cause trouble and shouldn't be available. Attempts to call them will likely have resulted from incompletely converted single-threaded code. Debugging will be far easier if such errors are caught at compile or link time than if they result in run time errors.

Strong arguments can also be made, however, for any new functions which are introduced specifically for multi-threaded programs also being available when building single-threaded programs; any functions which are useful in multi-threaded programs will also be useful in single-threaded programs since single-threaded programs are merely a special case of multi-threaded programs. Moreover, having the new functions available for both types of programs further enhances the potential compatibilities between them.

Compilation Support for Fast Locking

The time needed to successfully acquire and release an uncontested lock defines a lower bound on the time needed for a thread to access any shared data structure in a multi-threaded program. Since this lower bound will be a limiting factor for many programs, it is important that it be as small as possible.

If procedures are called to acquire and release locks, then the lock acquire/release bound will be at least two procedure call times. On many machines, this may be unacceptably high. Several methods may be used to eliminate these procedure calls:

- **Use of locking macros:** While lock acquisition typically requires a special instruction to be executed, lock release often only requires a simple store. On such machines, the lock release procedure call can be eliminated

through the use of a macro.

- **Preprocessing assembler input:** Most C compilers will generate assembler input which can be transformed by other tools before assembly. Lock calls can be eliminated from the generated assembler input by substituting equivalent inline expansions. (While less common for applications code, this technique has a long history of use in building kernels.)
- **Compiler locking support:** Some compilers have support for directly generating machine-specific locking instructions. When such support is available it can be used to optimize the generated locking code.

Note that any machine-specific code can be embedded in macros or tools provided through machine-independent interfaces. Thus, while all these techniques rely on machine-specific features, each can still be used by portable programs.

Compatibility Between Single and Multiple Threaded Code

Source Compatibility

A large degree of source compatibility is possible between code using single-threaded and multi-threaded versions of C libraries. Many library functions are already reentrant and need no special treatment. Functions maintaining persistent state can use internal locking, allowing for source compatibility between single and multiple threaded calls. If *errno* is implemented as a thread-specific variable, then source compatibility can be maintained for functions which use it as well. In general, source compatibility is only inhibited when functions are used which have non-reentrant interfaces; on systems which allocate former statics as thread-specific data, nearly perfect source compatibility is achievable.

Object Compatibility

A certain degree of object compatibility is possible between code using single-threaded and multi-threaded versions of C libraries. Any previously reentrant functions will not have changed, preserving object compatibility. Likewise, if internal locking is done by multi-threaded library functions, then old single-threaded main programs may sometimes be successfully linked against them. For instance, an old single-threaded program might be linked with a new multi-threaded quicksort routine, accomplishing a substantial speedup for very little effort. Most exported library data structures will not need to have locks added and so can remain unchanged.

Some library data structures do need to have locks added, but these can often be added in upward-compatible ways. In particular, while a recursive (counted) mutex lock must logically be added to every stdio "FILE" buffer, the locks can actually be allocated in a parallel data structure.

This preserves the layout of the original "FILE" structure, allowing single-threaded objects using stdio functions to call new multi-threaded objects also using them, and allowing some single-threaded objects which use stdio functions to be called by multi-threaded programs (provided that the multi-threaded code performs explicit *flockfile()* and *funlockfile()* calls around uses of the old objects). Of course, a performance price will be paid for this type of object compatibility; finding locks in parallel data structures will almost certainly be slower than accessing them directly in expanded "FILE" structures.

Object compatibility for functions with non-reentrant interfaces is achievable when a common compilation/execution environment is used for both single and multi-threaded programs. Such objects can be compatible when both the single and multi-threaded programs actually allocate logically static return values in thread-specific storage.

One other trick (a.k.a. "hack") may be performed to enhance object compatibility. Assuming that *errno* is allocated in thread-specific storage and is accessed via a clever macro or function, a global int named *errno* can still be present as well. Multi-threaded versions of library functions can then both set the thread-specific and global *errno* values whenever an *errno* value is returned. This allows the multi-threaded version of such a function to be called by single-threaded code which expects the *errno* value in the global.

Approaches Being Taken by the UNIX Community

A number of different implementations of multi-threaded libraries have been built by research and industry groups. Several more are proposed or under construction. Some of those which already exist have been built by Mach, Encore, Apollo, Chorus, Convex, and the Open Software Foundation. The OSF has implemented the draft POSIX Threads proposal (P1003.4a a.k.a. "pthreads") which includes reentrant libraries. Implementations of a UNIX International threads requirements document which specifies reentrant libraries are under way by UNIX System Laboratories (USL, formerly the UNIX Software Operation of AT&T), Sun and other member companies. Substantial numbers of multi-threaded library implementations are appearing. Happily to a large extent, many of these implementations agree on the approaches taken.

A more detailed presentation of some of the particular choices made is presented below. While not intended to be comprehensive, the choices presented are intended to be indicative of some of the directions that the industry has taken.

- **Cache Synchronization Techniques:** All known implementations of multi-threaded C libraries perform any necessary cache

synchronization operations automatically.

- **errno:** Unanimous agreement seems evident within the UNIX community that *errno* should be maintained on a per-thread basis. Several implementations such as those by Encore, Chorus, and USL also store into a global *errno* value.
- **The stdio Functions:** All multi-threaded implementations use internal locking for the stdio functions. Most, including Mach, pthreads/OSF, UI/USL, Sun, Convex, and Chorus have taken the "safe allowing fast" approach towards the stdio macros (e.g., *putc()*). Most also provide a *flockfile()*-like primitive permitting explicit locking control.
- **Functions with Non-Reentrant Interfaces:** Two different approaches have been adopted for functions with non-reentrant interfaces. Chorus and Convex have chosen to maintain such state in thread-specific storage. (Chorus does this in software; Convex does this with virtual memory.) Others such as pthreads/OSF and UI/USL have chosen to provide alternate reentrant versions of such functions.
- **Functions which Maintain State Across Invocations:** All multi-threaded implementations of *malloc()*, *free()*, etc. use internal locking. Beyond this agreement, the same two approaches have been taken for functions which maintain state across invocations as were taken for functions with non-reentrant interfaces.
- **Signals:** Signal treatment is an area of divergence. Mach and OSF/1 have per-process signals. Chorus has broadcast to "interested" threads signals. Encore has several implementations. Pthreads is going to ballot with per-thread singly delivered signals.
- **Thread-Specific Data:** The Mach C-Threads package provides a single thread-specific data variable. Chorus uses a key-value scheme. Convex uses architectural support for thread-private memory. Sun provides compiler support for a small set of thread-private data locations. Pthreads provides a general key-value scheme with destructors.
- **Orderly Cancellation Mechanism:** The pthreads proposal contains cancellation interfaces which can be layered onto existing facilities. (This was modeled after the Alert facility used in Modula-2+.)
- **Compilation Environment:** The two main approaches taken towards compilation environments have been to provide a single compilation environment, and to provide a conditional compilation environment. Apollo and Chorus have both adopted single unified compilation environments. Mach, OSF, and USL have all adopted conditional compilation

environments.

- **Compilation Locking Support:** Mach uses both locking macros and (optional) preprocessing of assembler input to eliminate procedure calls for locking. Convex has compiler support for locking and parallel code generation. UI/USL plans to support inline lock expansion on some architectures.
- **Compatibility Between Single and Multiple Threaded Code:** Convex and Chorus achieve nearly perfect compatibility between single and multi-threaded code through the use of thread-specific data. Pthreads/OSF and UI/USL, on the other hand, both took the position that using thread-specific data to approximate perfect source compatibility represented unnecessary and non-obvious mechanism, and that multi-threaded programs would be better served by using new reentrant functions instead.

Conclusions

UNIX-like systems supporting multi-threaded programs are becoming increasingly common. Both parallel programming requirements and the advantages of a synchronous programming paradigm are driving the industry in this direction.

As systems supporting multi-threaded programs become more common, increasing numbers of applications are being written as multi-threaded programs. To the extent that programmers can capitalize on existing C library interfaces and implementations when writing multi-threaded code, such code will be both easier to write and easier to understand.

While migrating the C libraries to be usable in multi-threaded programs allows for a wide range of possible design choices, some alternatives have been shown to have clear advantages. These advantages have been recognized and utilized by many of the multi-threaded C library implementations which have appeared to date.

While clearly many details differ between the existing implementations, many of the core elements, such as using internal locking whenever possible, do agree. Measured in terms of the similarity of commonly used routines, they typically have far more in common than they have differences. There has been substantial industry convergence around the important issue of multi-threaded C libraries.

Nonetheless, a number of details are different between the many implementations. In the long run, the industry will not be well served by numerous interfaces which are similar in spirit but incompatible in practice. Collaboration by a wide range of the industry on the POSIX pthreads effort provides one possible sign that convergence will continue.

Finally, of course, multi-threaded libraries are not an end in and of themselves. Nonetheless, they will provide (and are already providing) an important basic tool for utilizing parallel programming to accomplish real-world tasks.

Acknowledgments

A number of individuals deserve special mention for having contributed to my understanding of the issues presented in this paper. In particular, Eric Cooper (of CMU), Bill Cox (of USL), Rick Greer (of Interactive), Bob Knighten (of Encore), Simon Patience (of OSF), and Garret Swart (of DEC SRC) have all participated in long (and sometimes actually enjoyable!) sessions where we tried to look at all the alternatives. Trish Jones (my wife!) also deserves special mention both for her technical writing help and her patience.

References

- [1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. Summer 1986 USENIX Technical Conference and Exhibition*. June, 1986.
- [2] American National Standards Institute. *American National Standard for Information Systems, Programming Language, C*. American National Standards Institute, Inc., 1990. ANSI X3.159-1989.
- [3] Francois Armand, Frederic Herrmann, Jim Lipkis, and Marc Rozier. Multi-threaded Processes in Chorus/MIX. In *Proc. of EUUG Spring'90 Conference*, pages 1-13. Munich, Germany, April, 1990. Chorus systemes Technical Report CS/TR-89-37.3.
- [4] *Unix System V Release 4.0 Programmer's Reference Manual* AT&T, 1989.
- [5] Eric C. Cooper, Richard P. Draves. *C Threads*. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie Mellon University, June, 1988.
- [6] Julie Kucera, CONVEX Computer Corporation. Making libc Suitable for Use by Parallel Programs. In *First USENIX/SERC Workshop on Experiences with Building Distributed and Multiprocessor Systems*. October, 1989.
- [7] Paul R. McJones, Garret F. Swart. *Evolving the UNIX System Interface to Support Multithreaded Programs*. Technical Report Research Report 21, Digital Systems Research Center, September, 1987.
- [8] Open Software Foundation. *Application Environment Specification (AES), Operating System Programming Interfaces Volume*. Prentice Hall, Inc., 1990. ISBN 0-13-043522-8.
- [9] *Threads Extension for Portable Operating Systems*. Technical Committee on Operating Systems of the IEEE Computer Society for Work Item: JTC1.22.21.2, 1990. Draft P1003.4a/D5.
- [10] UNIX International. *Multiprocessing Work Group, Final Report*. UNIX International, Inc., 1989. Draft of 24 October 1989.

Michael Jones is currently pursuing his Ph.D. in Computer Science at Carnegie Mellon University. His interests include operating systems, parallel programming and his fellow human beings. He implemented the multi-thread safe versions of the stdio functions for the Mach project upon which the OSF and Encore



libraries are based. He has also worked on the Mach C-Threads implementation and the Mach Kernelization project. He is an active participant in the POSIX pthreads effort and in the UNIX International Multiprocessing Working Group. His U.S. Mail address is Michael B. Jones; Carnegie Mellon University; Computer Science Department; Pittsburgh, PA 15213. His email address is Michael.Jones@cs.cmu.edu.

A Tree-Based Packet Routing Table for Berkeley Unix

Keith Sklower – University of California, Berkeley

ABSTRACT

Packet forwarding for OSI poses strong challenges for routing lookups: the algorithm must be able to efficiently accommodate variable length, and potentially very long addresses. The 4.3 Reno release of Berkeley UNIX uses a reduced radix tree to make decisions about forwarding packets.

This data structure is general enough to encompass protocol to link layer address translation such as the Address Resolution Protocol (ARP), and the End System to Intermediate System Protocol (ES-IS), and should apply to any hierarchical routing scheme, such as source and quality-of-service routing, or choosing between multiple Datakits on a single system.

The system uses a message oriented mechanism to communicate between the kernel and user processes to maintain the routing database, inform user processes of spontaneous events such as redirects, routing lookup failures, and suspected timeouts through gateways.

Introduction

An important focus of the 4.3 Reno release of Berkeley UNIX was to make support for the OSI protocols publicly available. OSI addresses are typically very long (20 bytes) and with the explosive growth of the Internet, a router may have to contend with thousands of them.

The traditional hash-based scheme of routing lookups would perform poorly in this environment. The older algorithm assumed that it would be cheap to compute hashes, that one could easily identify the network portion of an address, and easily compare them.

It is likely to be expensive to compute the hash of a 20 byte address. Moreover, where there are multiple hierarchies, it would be complicated and context dependent to identify which portion of the address should be considered as "the network portion" for comparison at changing levels. In general, it is not apparent how to accommodate hierarchies while using hashing, other than rehashing for each level of hierarchy possible.

Van Jacobsen, of the Lawrence Berkeley Laboratory, suggested using the PATRICIA algorithm (described below), but with an additional invariant to maintain a routing tree. This meshes extremely well with notions of multiple hierarchical defaults, and the cost of an entire lookup is approximately the same as the cost of computing a single hash.

Since there is now a means to store variable length addresses, and reason to use addresses of differing sizes within a given route (using a protocol destination address with a link-layer gateway to accomplish ARP-like translation, for example), it

was decided that using fixed length ioctl's to communicate between the kernel and routing process would be too restrictive. Instead, a message based mechanism is used for passing routing information to and from the kernel. This mechanism provides additional potential for remote management when future releases supply the ability to splice communications channels.

Kernel Issues

Routing Lookups

Restatement of the Problem.

Let's describe the problem once again in a little more detail: A packet arrives with a very long protocol address. If the destination address is not that of the local system, one wants to decide quickly how to forward it. This decision entails choosing a network interface and a next-hop agent. (Point-to-point links only have one agent on the other end of the link, so sometimes it's enough just to figure out which link!).

Some of the routing protocols currently in use give us criteria for making this choice, in what may seem a bizarre way: the space of addresses is partitioned into a set of equivalence classes by specifying a pair consisting of a prototype address and a bit-mask; a test address is deemed to belong to the class if any bit in which it differs from the prototype address corresponds to a zero bit of the provided mask.

Let's give an example, using the protocol addresses for the Internet Family [Post], which are 32-bit numbers:

Example 1: Some Address Classes

Prototype	Mask	ClassName
0x80030000	0xffff0000	LBL
0x80200000	0xffff0000	Berkeley
0x80208200	0xfffff00	CsDivSubnet
0x80209600	0xfffff00	SpurSubnet
0	0	TheOutside

The author's machine (okeeffe.Berkeley.EDU) has the address 0x80208203. Consequently, it belongs to the classes Berkeley, CsDivSubnet, TheOutside, but not LBL nor SpurSubnet. With each class is associated a networking interface, in most cases a next-hop agent, and a collection of other useful information, and that collection is referred to as a "route". Continuing the example, okeeffe.Berkeley.EDU can talk directly to any system in the class CsDivSubnet (all such systems are on a single ethernet), but requires an intermediary to talk to anybody else.

The routing lookup problem is to find the most specific class containing a given protocol address. Paradoxically, that will be the one with largest number of one bits in the mask. The NSF net may provide a regional router with about 2000 routes of this type. The lookup algorithm must look up the

appropriate class quickly (among both numerous and lengthy addresses), and yet have nice properties with respect to masks.

The algorithm

The collection of prototype addresses are assembled into a variant of a PATRICIA tree, which is technically a binary radix tree with one-way branching removed. (In fact some writers call any tree with explicit external and internal nodes a *trie*). Although this algorithm is given a lengthy exposition in [Sedg] and also is discussed in [Knut] and [Morr], we will review it here.

We build a tree with internal nodes and leaves. The leaves will represent address classes, and will contain information common to all possible destinations in each class. As such, there will be at least a mask and prototype address. Each internal node represents a bit position to test. Given the tree and a candidate address thought of as a sequence of bits, the lookup algorithm is as follows:

1. Set node to the top of the tree.
2. If at a leaf node, stop.
3. Extract a bit position to test.
4. If that bit of the candidate address is on, set the node to the right child of the current node,

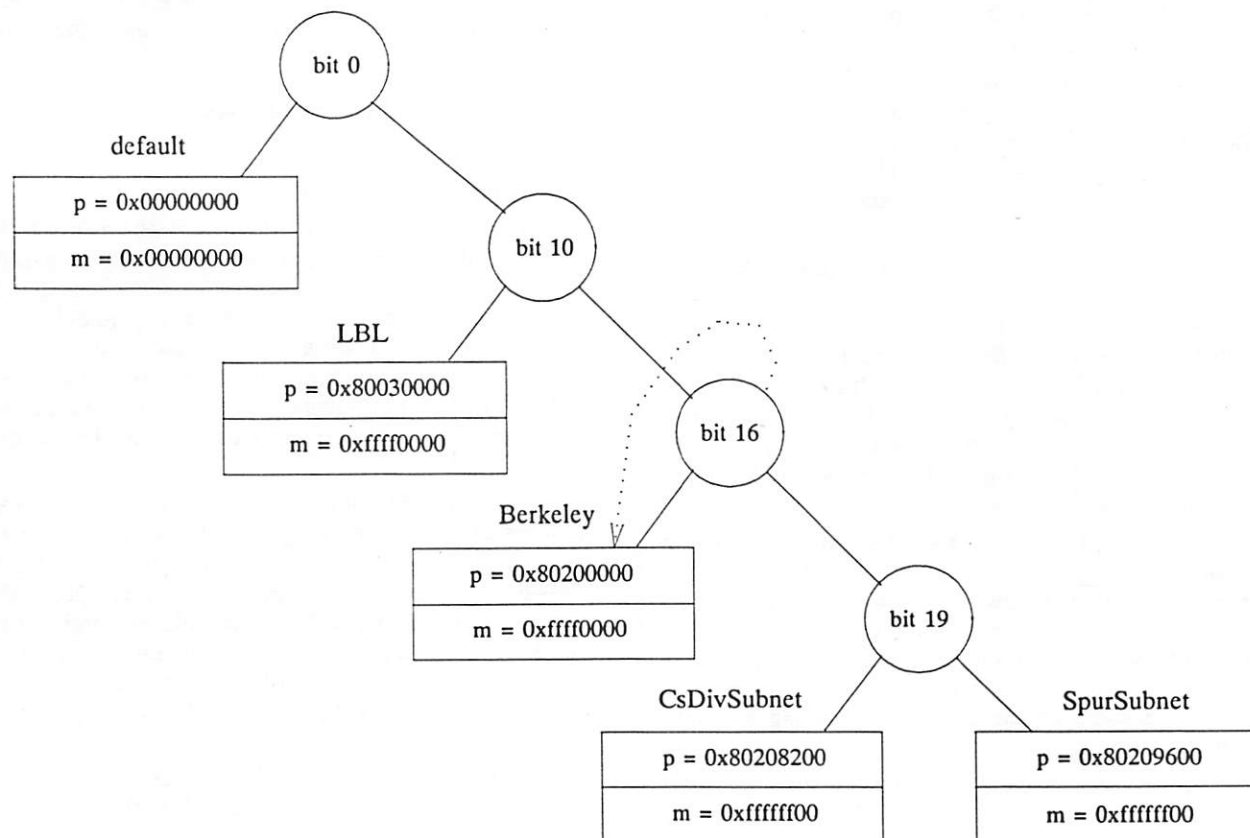


Figure 1

otherwise set node to the left child.

5. Repeat steps 2 - 4.

Once we arrive at a leaf node, we need to check whether we have selected the appropriate class. The class may consist of a single host. This is a special case where the mask consists of all one bits, but it is a common enough occurrence that we check for it (by use of a null pointer for the mask), and do an outright string compare. Otherwise, in which case we perform the masking operation.

It is possible to have the same prototype address with differing masks; this is handled by a linked list of leaf nodes. This arises due to boundary conditions for the smallest representation of the default route (which collides with the boundary marker for the empty tree). It also arises if you want to route to subnet 0 of a subnetted class A or B internet address.

If the leaf node isn't correct, then we backtrack up the tree looking for indications that a more general mask may apply (i.e. one having fewer one bits). This may happen if we are asked to look up an address other than the prototype addresses used to construct the tree. Rather than keep a separate stack of nodes traversed while searching the tree, backtracking is facilitated by having explicit parent pointers in each node. This also facilitates deletion, and allows non-recursive walks of the tree.

A Lookup Example

Figure 1 shows how one would construct a reduced radix tree to decide among the prototype addresses given in the example of address classes. In examining the address for okeeffe.Berkeley.EDU (0x80208203), we find that bit 0 is on (0x80000000: go right), bit 10 is on (0x00200000: go right), bit 16 is on (0x00008000: go right), but that bit 19 is off (0x00001000: go left). And, in fact okeeffe does match the CsDivSubnet class.

If we were to look up another machine at Berkeley, say miro.Berkeley.EDU (0x80209514), we are driven down to the SpurSubnet class, which does not match. So we backtrack up to the second internal node above it, which has an indication that there is a mask which may apply. This is represented in the diagram by the dotted line, which actually points to data associated with mask contained in the leaf, rather than the leaf itself.¹

¹The internal node based on bit 10 does not have an indication that there is a mask which may apply to it. This because any search backtracking through there would have had to had a 1 for bit 10 (since it otherwise would have been trapped by the leaf for LBL itself), as the LBL class has that bit off. There is a measure of how high in the tree a mask can apply (in our current scheme) which we call the index of the mask. The interested reader can peruse the source code in 4.3 Reno for further elaboration.

Backtracking only occurs when given packets are covered by a default route, or when non-prefix masks are employed. The current implementation deals with non-contiguous masks in a way requiring an explicit masking and re-lookup operation for each possibly applicable mask encountered while backtracking. This has the advantage that routes can be entered one by one without requiring searches or reorganization of subtrees.

Researchers in the field ([Tsuc], [Butl]) have suggested this might be avoided by constructing a tree in which the nodes test bits in non-increasing order, governed by the masks found in leaves underneath. This is the object of current study.

Comparison with the previous method.

Releases of Berkeley UNIX prior to 4.3 Reno employed an explicit three level hierarchy for routes, routing first to hosts, then to networks, then to defaults. The collections of host routes and network routes were entirely separate hash arrays.

Given a candidate address, an address family specific method would be invoked to compute a hash value for the hosts array, and a bucket chosen. Each element of the bucket would be compared against the candidate address, via a second address family specific method for each comparison (i.e. requiring a subroutine call per comparison).

If the candidate address was not found, the process would be repeated with the network hash array. If that failed, a list of defaults would be searched to see if there were any for the address family of the candidate address.

By contrast, the initial search of the tree also is written in a protocol independent way. Furthermore, the new algorithm performs its comparisons in a protocol independent way, permitting the back-tracking loop to occur without separate subroutine calls.

It is interesting to note that in the average case, PATRICIA trees are approximately balanced. The expected length of a search is only $1.44 \log$ (number of entries); and of course the maximum possible search is the number of bits in the address. By contrast, the worst case for a degenerate hash is the number of entries to be searched. So, if we had 2000 IP entries, in the PATRICIA case one would expect 15 bit tests, whereas in the typical hashing situation of \sqrt{N} , one would expect about 44 hash entries in 44 hash chains, with an average of 22 comparisons after hashing. The (pathological) worst case is 32 bit tests versus 2000 compares.

The Routing Entry

The routing entry is a collection of information for use by protocol implementations. It has a base part which contains all the aspects common to a class of hosts with which we might want to

communicate that we can express in a protocol-independent (or protocol-uniform) way. It certainly includes the connectors used in constructing the tree, the prototype address (which we think of as the destination address), and the mask.

There are binary flags present which may greatly alter the interpretation of the route, or even cause new ones to spring into existence. (see **Cloning Operation**, below). There are pointers to a protocol independent control structure describing the network interface (the *ifnet* structure), and the protocol level address which should be used in identifying the local system (the *ifaddr* structure).

There is a possible gateway address, that is used in situations in which protocol packets must be sent to the intended recipients via an intermediary. This mode of operation is identified by having a flag **RTF_GATEWAY**. In the other case (where no gateway is required), there is a pointer to device- and protocol- specific information, such as link-layer to protocol address translations, or even other protocol control blocks for situations such as running connectionless protocols over X.25.

The route includes a collection of statistics that are commonly maintained by reliable and flow controlled protocols, such as round-trip time, round-trip time variance, maximum packet size for this path, maximum number of gatewaying or forwarding operations expected, in-bound and out-bound throughput measures, and a bitmask to indicate if any of these values should be left unaltered by protocol operation. There are some other statistics that are purely housekeeping matters, such as the number of protocol control blocks keeping a reference to this route.

Cloning Operation

As mentioned above, it is sometimes convenient for skeletal routing entries to be created and partially filled in upon first reference (or lookup), with the missing information to be supplied later. Allocating a dedicated routing entry at initial connect time saves the expensive of checking validity on each use.

An example of this would be a user opening a TCP connection to another machine on the same ethernet, for which the link-layer address was not yet known. The creation of such entries would triggered by the flag **RTF_CLONING** in a route being looked up.

For other sorts of link layer translations, such as IP address to X.121 addresses for use over a public data network, it may be desirable to have a message sent to a user level daemon when the route is created, requesting an external resolution of protocol addresses. This mode is enabled by the flag **RTF_XRESOLVE**.

Another example would be a configuration in which there are many different subnets on the other side of a serial link, where each subnet may have different performance characteristics (which could be learned operationally), but that each use would be infrequent and random enough that it would be wasteful permanently to allocate space for routing entries to each possible subnet in advance.

Here, a way to specify the netmask for the newly cloned route is necessary, which needs to be more specific than the netmask for the cloning route which creates it. Thus, the route structure includes a pointer for this secondary mask, which is only used in such a situation. The primary netmask is used for "trapping" the lookup; the secondary mask would be used as the primary mask in the newly created route which would restrict additional lookups to that newly identified class of hosts.

Black Holes (or Border Patrol)

A handy use for hierarchical defaults would be at the gateway of a campus to catch packets for non-existent subnets or hosts within the campus that would otherwise be sent to the default route advertised by the regional connection to a backbone network. This is easily implemented by the flag **RTF_REJECT**.

In 4.3 Reno, the network output routines added an additional parameter, a pointer to the route. This parameter enabled cached link layer information to be retrieved, but also allows the loopback driver to recognize the **RTF_REJECT** flag. When it does so, it consumes the offending packet and returns **EHOSTUNREACH** or **ENETUNREACH**, prompting the protocols to do the appropriate magic with no other changes.

Ancillary addressing structures:

The protocol independent network interface structure (ifnet)

For each network interface device, there is a structure describing a number of protocol independent elements. Some of these serve to identify the device: a printable name and a unit identifier. There are also general statistics kept about each device, some inherent about the device itself (such as maximum packet size, a generic type for the device, and binary flags indicating whether the device is point-to-point or broadcast or deaf to its own broadcasts). There are other statistics reflecting use, such as number of packets in and outbound, (and how many of those encountered errors), time of last use, total number of bytes transmitted and received. There are a collection of methods associated with the device. These include a general output routine to process packets and place them on an output queue, an internal routine to initiate transmission, an ioctl routine, initialization, reset, and two routines

used at startup time.

This structure has escaped relatively unchanged from previous versions of BSD; a good description of it can be found in [Leff]. The new additions include the device start method which makes it possible for all ethernet drivers to use a common output routine, more statistics required by SNMP [Case], and throughput statistics used for protocol operation.

The protocol address structure (sockaddr)

All protocol addresses have a common two byte header detailing the length and type of the address.

The 4.3 Reno release adds a device independent link-layer address format, which may be used in sending link-layer packets or disambiguating interfaces when more than one have the same protocol addresses.

The protocol dependent interface addressing structure (ifaddr)

There may be multiple protocol addresses associated with each network interface. The *ifaddr* structure provides a place to store them and other device- and protocol-specific information. In fact, some protocols allow either multiple names for the same interface, or the same name for multiple interfaces or both.

Even though the values will differ from protocol to protocol, there are some other common elements that can be identified, so that this structure has a protocol independent header, with a protocol specific tail expected to follow immediately.

The protocol independent elements include the address, associated subnet mask, destination or broadcast address, linkage to the next address and the associated *ifnet*, a method to be invoked when routes associated with this address are created or deleted, a routing entry associated with this address for this interface, and generic flags for this level.

This structure is also discussed in [Leff]. The method, routing entry, and flags fields are new (since 4.3 BSD), and the protocol addresses have been changed to pointers rather than allocating fixed size spaces for them.

Messages and Formats

As mentioned above, BSD has adopted a message passing approach for management of the routing table, for a variety of reasons. First, network address are of variable length, and we may have varying numbers of them in differing operations. Second, it provides a clean and uniform way of informing a routing process of spontaneous events, such as redirects, routing misses, requests to resolve link layer address translations, or internal evidence

that a gateway may have crashed, (due to lack of acknowledgments across a class of connections). Third, it provides a way for making additions or changes to the management interface while maintaining backwards compatibility. A version number is embedded in the message header, and each message is self delimiting, so that any unknown message to the user program can be skipped. Finally, for the future when it will be possible to splice message streams together, it provides an easy path towards remote system management.

The basic format

All messages have a common header, and some varying number of protocol addresses appended to them. The header includes the total length of the message, a version number and a type, which allows non-understood messages to be skipped. There is space for a user-supplied sequence number. The returned message includes the pid of the originating process.

The header also includes a number of metrics, a bit mask to identify which are being specified, and a second bit mask to specify which metrics must remain unchanged by the protocols.

The interpretation and number of the trailing protocol addresses is specified by a bitmask. The potential addresses are:

Symbolic Name	Description
RTA_DST	Prototype address
RTA_NETMASK	Bitmask for describing class
RTA_GATEWAY	Gateway
RTA_GENMASK	Bitmask for routes created by cloning
RTA_IFA	The protocol address to be used as a source address when sending to hosts covered by this route
RTA_IFP	An address unambiguously specifying which interface (struct <i>ifnet</i>) is associated with this route, such as a link-level address.
RTA_AUTHOR	Address identifying sender of redirect, etc.

Message Types.

In this section, we'll discuss each of the message types, describing features unique to each, and contrasting the intent of otherwise similar looking messages.

RTM_ADD – Enter a new route into the table.

This is the basic operation for creating the routing table. The destination and gateway addresses must be present. If there is no netmask present, the route is assumed to be a route to a host. In the case where the host or class specified by the route is directly reachable, the gateway address may be used to specify a link layer address (for hosts), or the protocol address of the outgoing interface, which may implicitly identify the *ifaddr* and *ifnet* structure

pointers. Even the case of a class reached via a gateway, one may be able to deduce the interface from the address of the gateway. If there is ambiguity about this, as may be the case in OSI protocol operation, they must be explicitly supplied.

The flags may specify cloning operation, as described in section 2.2.1. If the the new routes are to specify a subclass instead of a host route, a generating bit mask needs to be supplied.

RTM_DELETE – Remove an entry from the table

If there is only one entry in the routing table with a given prototype address, that is sufficient to identify the route to be deleted. Otherwise, the netmask associated with the route must additionally be specified.

RTM_CHANGE – Alter characteristics of a route

Due to gateways going up or down, it may be desirable to change the designated forwarding agent for a class of hosts. It is also desirable to do so atomically (locking out forwarding requests), so that there isn't a period in which incorrect host or network unreachable protocol messages are generated in response to packets to be forwarded. Changing the gateway implicitly or explicitly requires changing the associated *ifaddr* and *ifnet* structures.

In this message, one can also alter the metrics associated with a route or some of the flags (cloning, resolving, link-layer-ness).

Altering the netmask associated with a route is not permitted, since this would affect the geometry of the tree; instead one deletes and re-inserts.

RTM_GET – Look up route and report characteristics.

This message is diagnostic in nature. The user supplies a destination and the best match route indication is returned, along with all of the metrics filled in. Where there are multiple routes with the same prototype address (but multiple netmasks), specifying the netmask will allow the user to select the appropriate route.

RTM_REDIRECT – Request to change gateway.

This message is an example of a spontaneous event. Both the TCP/IP and OSI family of protocols have the potential for receiving an advisory report from a gateway that the initiating system would be better off sending a packet to another gateway on the same network for forwarding to a remote location.

When the routing table is maintained by a user-level process, it is important that the routing process be notified of any changes to the routing table.

For OSI protocols, the initiating system may get a message specifying the original destination, a bitmask specifying a class of hosts for which this redirect also pertains, the replacement gateway to be used, and the author of the message.

RTM_LOSING – Trouble reports.

“Reliable” byte and message stream protocols such as TCP or OSI-TP keep retransmission timers. If a connection suddenly stops working, it may signal the loss of a gateway. User-level routing processes may be interested in keeping track of such events, at the very least to determine if it appears the local or a remote gateway as failed. This message identifies the route which covers the remote hosts involved in such lossage.

RTM_MISS – A routing table lookup failed.

The local system was asked to forward a packet or initiate a connection to a destination for which it could not find a suitable route. One could imagine a system attached to a wide area network which would only allow a limited number of active reachable destinations, such as an X.25 network. The system might only enter those active peers in the network table, and open new ones (or close old ones) based on the number of misses.

This may be useful for purely diagnostic purposes as well.

RTM_RESOLVE – Request to complete route info via CHANGE.

This is very similar to the RTM_MISS message. It is intended for cloning operation (which would not otherwise cause an RTM_MISS type message) where some information needs to be obtained externally from some process that is not convenient to be coded directly into the kernel.

Measurements

We performed a synthetic test of constructing a routing table of about 1600 entries using both the new and old methods (in a user-level process). We then searched each table randomly 100000 times for entries in the table. The routing table was constructed from data obtained on a gateway system at Cornell University, which stands between the Cornell campus and the NSF net.

In fact, the time required to construct a table of 1600 routes was on the order of half a second for either method; our test actually measured constructing the table 10 times and emptying it 9 times. The test results show the new (radix tree based) method to be about 50% faster in constructing the tree and 200% faster in searching it. The overhead column represents the time required to loop through all routes calling a routine that does nothing instead of

adding, deleting or lookup a route. The units in the table below are user time in seconds, as measured on a CCI tahoe processor running 4.3 Reno BSD.

operation	old	new	overhead
create	10.28	6.75	.10
search	29.72	7.38	.86

References:

- [Butl] Butler, Duane M. Private communication, August 1990.
- [Case] Case, J.D., *et al.* Simple Network Management Protocol RFC 1157, SRI Network Information Center, May 1990.
- [Knut] Knuth, Donald E. *The Art of Computer Programming*, Vol 3. pp. 490-493, Addison-Wesley, Reading MA 1973
- [Morr] Morrison, Donald R. *J ACM* 15 (1968), pp. 514-534
- [Leff] Leffler, Samuel J. *et al.* *The Design and implementation of the 4.3BSD UNIX® Operating System*. Addison-Wesley, Reading, MA 1988
- [Post] Postel, J. "Internet Protocol - DARPA Internet Program Protocol Specification," RTC 791, USC Information Sciences Institute, September 1981.
- [Sedg] Sedgwick, Robert. *Algorithms in C*. pp. 253-257, Addison-Wesley, Reading MA 1990
- [Tsuc] Tsuchiya, Paul. Efficient assignment of addresses in the Internet. (IETF Proceedings, July 1990),

Keith Sklower graduated in 1972 from Reed College in Portland Oregon with a B.A. in Mathematics and Physics. After doing some graduate work in Math at U. C. Berkeley, he has worked for the Computer Science Department there since 1978, initially helping to develop and maintain the Franz Lisp system, but also implemented support for XNS protocols in 4.3 BSD. He currently is integrating support for OSI protocols for the next release of Berkeley UNIX.



Appendix A. Radix Tree Declarations and Search Algorithm

We include a somewhat simplified version of the header file for the radix tree; but the algorithm for searching the tree is taken verbatim.

```

/*
 * Copyright (c) 1988, 1990 Regents of the University of California.
 * All rights reserved.
 *
 *      @(#)radix.h      7.4a (Berkeley) 11/28/90
 */

/*
 * Common Indices
 */
struct radix_info {
    short    ri_b;           /* bit offset; -1-index(netmask) */
    char     ri_bmask;      /* node: mask for bit test */
    u_char   ri_flags;      /* enumerated next */
}
#define RNF_ROOT          1      /* leaf is root leaf for tree */
#define RNF_ACTIVE        2      /* This node is alive (for rtfree) */

/*
 * Radix search tree node layout.
 */

struct Radix_node {
    struct    radix_mask *rn_mklist; /* indication a mask may apply */
    struct    radix_node *rn_p;      /* parent */
    struct    radix_info rn_ri;      /* bit number and mask, flags */
    int       rn_off;               /* precomputed offset for byte test */
    struct    radix_node *rn_l;      /* progeny */
    struct    radix_node *rn_r;      /* progeny */
};

struct Radix_leaf {
    struct    radix_mask *rn_mklist; /* our handle to the annotation */
    struct    radix_node *rn_p;      /* parent */
    struct    radix_info rn_ri;      /* bit number and mask, flags */
    caddr_t   rn_key;               /* object of search */
    caddr_t   rn_mask;              /* netmask, if present */
    struct    radix_node *rn_dupedkey;
};

/*
 * The actual radix node struct is defined
 * in terms of a structure containing a union with copious defines such as:
 */
#define rn_key rn_u.rn_leaf.rn_Key
#define rn_b   rn_ri.ri_b

```

```

/*
 * Annotations to tree concerning potential routes applying to subtrees.
 */

extern struct radix_mask {
    struct    radix_info rm_ri;           /* bit number and mask, flags */
    struct    radix_mask *rm_mklist;     /* more masks to try */
    caddr_t   rm_mask;                   /* the mask */
    int       rm_refs;                   /* # of references to this struct */
} *rn_mkfreelist;

struct radix_node *
rn_search(v, head)
    struct radix_node *head;
    register caddr_t v;
{
    register struct radix_node *x;

    for (x = head; x->rn_b >= 0;) {
        if (x->rn_bmask & v[x->rn_off])
            x = x->rn_r;
        else
            x = x->rn_l;
    }
    return x;
};

```


Appendix B: Header Files for routing messages, structures.

This also is a slightly simplified version of the actual header file:

```

/*
 * Copyright (c) 1980, 1990 Regents of the University of California.
 * All rights reserved.
 *
 *      @(#)route.h      7.12a (Berkeley) 11/28/90
 */

/*
 * These numbers are used by reliable protocols for determining
 * retransmission behavior and are included in the routing structure.
 */
struct rt_metrics {
    u_long    rmx_locks;        /* Kernel must leave these values alone */
    u_long    rmx_mtu;          /* MTU for this path */
    u_long    rmx_hopcount;     /* max hops expected */
    u_long    rmx_expire;       /* lifetime for route, e.g. redirect */
    u_long    rmx_recvpipe;     /* inbound delay-bandwidth product */
    u_long    rmx_sendpipe;     /* outbound delay-bandwidth product */
    u_long    rmx_ssthresh;     /* outbound gateway buffer limit */
    u_long    rmx_rtt;          /* estimated round trip time */
    u_long    rmx_rttvar;       /* estimated rtt variance */
};
/*
 * Bits for locking and initializing metrics
 */
#define RTV_MTU          0x1    /* init or lock _mtu */
#define RTV_HOPCOUNT   0x2    /* init or lock _hopcount */
#define RTV_EXPIRE      0x4    /* init or lock _hopcount */
#define RTV_RPIPE       0x8    /* init or lock _recvpipe */
#define RTV_SPIPE       0x10   /* init or lock _sendpipe */
#define RTV_SSTHRESH    0x20   /* init or lock _ssthresh */
#define RTV_RTT         0x40   /* init or lock _rtt */
#define RTV_RTTVAR      0x80   /* init or lock _rttvar */

struct rtentry {
    struct    radix_node rt_nodes[2];    /* tree glue, and other values */
#define rt_key(r)      ((struct sockaddr *)((r)->rt_nodes->rn_key))
#define rt_mask(r)     ((struct sockaddr *)((r)->rt_nodes->rn_mask))
    struct    sockaddr *rt_gateway;      /* value */
    short     rt_flags;                  /* up/down?, host/net */
    short     rt_refcnt;                 /* # held references */
    u_long    rt_use;                    /* raw # packets forwarded */
    struct    ifnet *rt_ifp;             /* the answer: interface to use */
    struct    ifaddr *rt_ifa;            /* the answer: interface to use */
    struct    sockaddr *rt_genmask;      /* for generation of cloned routes */
    caddr_t   rt_llinfo;                 /* pointer to link level info cache */
    struct    rt_metrics rt_rmx;         /* metrics used by rx'ing protocols */
    short     rt_idle;                   /* easy to tell llayer still live */
};
/*
 * Flags
 */
#define RTF_UP          0x1    /* route useable */
#define RTF_GATEWAY     0x2    /* destination is a gateway */

```

```

#define RTF_HOST          0x4          /* host entry (net otherwise) */
#define RTF_REJECT        0x8          /* host or net unreachable */
#define RTF_DYNAMIC       0x10         /* created dynamically (by redirect) */
#define RTF_MODIFIED      0x20         /* modified dynamically (by redirect) */
#define RTF_DONE          0x40         /* message confirmed */
#define RTF_MASK          0x80         /* subnet mask present */
#define RTF_CLONING       0x100        /* generate new routes on use */
#define RTF_XRESOLVE      0x200        /* external daemon resolves name */
#define RTF_LLINFO        0x400        /* generated by ARP or ESIS */
/*
 * Structures for routing messages.
 */
struct rt_msghdr {
    u_short    rtm_msglen;          /* to skip over non-understood messages */
    u_char     rtm_version;         /* future binary compatability */
    u_char     rtm_type;            /* message type */
    u_short    rtm_index;           /* index for associated ifp */
    pid_t      rtm_pid;             /* identify sender */
    int        rtm_addrs;           /* bitmask identifying sockaddrs in msg */
    int        rtm_seq;             /* for sender to identify action */
    int        rtm_errno;           /* why failed */
    int        rtm_flags;           /* flags, incl. kern & message, e.g. DONE */
    int        rtm_use;             /* from rtentry */
    u_long     rtm_inits;           /* which metrics we are initializing */
    struct     rt_metrics rtm_rmx; /* metrics themselves */
};
/*
 * Message Types
 */
#define RTM_ADD            0x1         /* Add Route */
#define RTM_DELETE        0x2         /* Delete Route */
#define RTM_CHANGE        0x3         /* Change Metrics or flags */
#define RTM_GET           0x4         /* Report Metrics */
#define RTM_LOSING        0x5         /* Kernel Suspects Partitioning */
#define RTM_REDIRECT      0x6         /* Told to use different route */
#define RTM_MISS          0x7         /* Lookup failed on this address */
#define RTM_LOCK          0x8         /* fix specified metrics */
#define RTM_OLDADD        0x9         /* caused by SIOCADDRT */
#define RTM_OLDDEL        0xa         /* caused by SIOCDELRT */
#define RTM_RESOLVE       0xb         /* req to resolve dst to LL addr */
/*
 * Bits for identifying trailing or optional sockaddrs.
 */
#define RTA_DST            0x1         /* destination sockaddr present */
#define RTA_GATEWAY        0x2         /* gateway sockaddr present */
#define RTA_NETMASK        0x4         /* netmask sockaddr present */
#define RTA_GENMASK        0x8         /* cloning mask sockaddr present */
#define RTA_IFP            0x10        /* interface name sockaddr present */
#define RTA_IFA            0x20        /* interface addr sockaddr present */
#define RTA_AUTHOR         0x40        /* sockaddr for author of redirect */

```


An X11 Toolkit Based on the Tcl Language

John K. Ousterhout – University of California at Berkeley

ABSTRACT

This paper describes a new toolkit for X11 called Tk. The overall functions provided by Tk are similar to those of the standard toolkit Xt. However, Tk is implemented using Tcl, a lightweight interpretive command language. This means that Tk's functions are available not just from C code compiled into the application but also via Tcl commands issued dynamically while the application runs. Tcl commands are used for binding keystrokes and other events to application-specific actions, for creating and configuring widgets, and for dealing with geometry managers and the selection. The use of an interpretive language means that any aspect of the user interface may be changed dynamically while an application executes. It also means that many interesting applications can be created without writing any new C code, simply by writing Tcl scripts for existing applications. Furthermore, Tk provides a special `send` command that allows any Tk-based application to invoke Tcl commands in any other Tk-based application. `Send` allows applications to communicate in more powerful ways than a selection mechanism and makes it possible to replace monolithic applications with collections of reusable tools.

1. Introduction

Tk is a new toolkit for the X11 window system [10]. Like other X11 toolkits such as Xt [1] or the Andrew toolkit [9], Tk consists of a set of C library procedures intended to simplify the task of constructing windowing applications. The Tk library procedures, like those of other toolkits, serve two general purposes: framework and convenience. First, they provide a framework that allows applications to be built out of many small interface elements called *widgets* (e.g., buttons, scrollbars, menus, etc.). The toolkit's framework makes it possible to design widgets independently, compose them into interesting applications, and re-use them in many different situations without re-design. The second purpose of the toolkit is to provide ready-made solutions for the most common needs of windowing applications. For example, Tk includes a set of commonly used widgets plus procedures to make it easy to build new widgets. Using Tk, it is possible to build many interesting windowing applications by plugging together existing widgets. Many other applications can be built by constructing one or two new widget types and combining them with Tk's existing widgets.

Although Tk's overall purpose is similar to that of other toolkits, its implementation has the unusual property that it is based around the Tcl command language. Tcl is a simple interpretive programming language designed to be embedded in applications and to work cooperatively with C code in the applications [8]. Tcl programs can be created and executed dynamically, and all of the functionality of Tk (and of Tk-based applications) is available through

Tcl. This gives Tk a greater degree of flexibility, dynamic control, and power than other toolkits. For example, Tcl can be used to modify the entire widget configuration of an application at any time. New applications can be created by writing Tcl scripts for a windowing shell or for existing Tk-based applications; C code is needed only for creating new widget types or data structures.

The most important feature of Tk is that it allows different applications to work together in powerful ways. Tk provides a remote-procedure-call-like facility that allows any Tk-based application to invoke Tcl commands in any other Tk-based application. This results in more powerful communication than the traditional selection or cut buffer. Current windowing applications are forced by the lack of good communication to lump large amounts of functionality into a single application. Tk makes it possible to replace such monolithic applications with collections of smaller specialized applications that communicate with each other using Tcl commands. These smaller tools are often re-usable for other purposes, thereby resulting in more powerful windowing environments.

Tk and Tcl also simplify windowing environments by making a single run-time command language available everywhere. There is less need for application designers to invent special-purpose languages or protocols to handle particular situations: the application can just use Tcl. For example, Tcl serves as a user interface description language. It is also easier to build a new application because the application designer need only implement a few key primitive operations for the application; Tcl allows those primitives to be composed

with other primitives within the application or in other applications. Tcl also simplifies things for users. Instead of learning a different command language for each application, a user need only learn Tcl. The user will then be able to program any Tk-based application merely by learning the application-specific primitives provided by that application.

```
set a 1000
print foo; print bar
```

Figure 1: Simple Tcl commands consist of fields separated by white space. The first field is a command name and the additional fields are arguments for the command. Commands are separated by semi-colons or newlines.

```
set msg "Hello, world"
set x {a b {x1 x2}}
```

Figure 2: Double-quotes or nested curly braces may be used to delimit complex arguments in Tcl commands. Each of the above commands has three fields in all. If an argument is enclosed in braces then the contents of the braces are passed to the command without any further interpretation (newlines and semi-colons are not command separators and the substitutions described in Figures 3-5 are not performed). If an argument is enclosed in quotes, then the substitutions in Figures 3-5 are performed on its contents.

```
print $msg
if $i<2 {set j 43}
```

Figure 3: Dollar signs invoke variable substitution in Tcl commands: the dollar sign and variable name will be replaced with the value of the variable in the argument passed to the command.

```
print [list q r $x]
set msg [format "x is %s" $x]
```

Figure 4: Tcl commands may contain other commands enclosed in brackets. When this occurs, the nested command is executed and its result is substituted into the argument of the enclosing command, replacing the bracketed command.

```
set msg "\{ and \[ are special"
print Hello!\n
```

Figure 5: Backslashes prevent special interpretation of characters like braces and brackets in Tcl commands. Backslashes can also be used to insert control characters into commands, as in the second command above.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the Tcl language and how the Tcl interpreter is embedded in applications. Section 3 summarizes the framework Tk

provides for building widgets. Section 4 describes how widgets are constructed and manipulated in Tk. Section 5 demonstrates the advantages of Tk with a few examples of user interface programming. Section 6 describes how Tk allows applications to work together by sending Tcl commands to each other. Section 7 presents the current status of Tk along with some size and performance measurements. Section 8 compares Tk to other toolkits and Section 9 concludes.

2. Summary of Tcl

Tcl stands for "tool command language." My goal in developing Tcl was to make it easy to generate powerful command languages for interactive applications. Tcl is a library package written in C. It implements an interpreter for a simple programming language that provides variables, procedures, control constructs like `if` and `for`, arithmetic expressions, lists, strings, and other features. Tcl also allows applications to extend the generic command set with application-specific commands. An application need only implement a few basic Tcl commands related to the application; when these are combined with the Tcl library a fully-programmable command language results. The paragraphs below summarize a few of the key features of Tcl; see [4] and [8] for more information on Tcl and how it has been used.

The Tcl language has a simple syntax with features reminiscent of the UNIX shells, Lisp, and C. Figures 1-5 summarize the complete Tcl syntax. In their simplest form (Figure 1), Tcl commands are like shell commands: they contain one or more fields separated by white space; the first field is the name of a command and the other fields are arguments passed to the command. Unlike UNIX shell commands, Tcl commands return string values. The Tcl syntax includes additional features for specifying complex arguments, substituting variable values, and executing nested commands (see Figures 2-5).

Tcl is an *embedded language*: it is a library that is designed to be linked together with C applications as shown in Figure 6. The main loop of the application generates Tcl commands. This could happen in any of several ways, depending on the application. One way is to read commands from standard input; this results in a shell-like program. Another way, used by Tk, is to associate Tcl commands with X events such as button presses or keystrokes; when an X event occurs, the corresponding commands are executed. When the application has generated a Tcl command it passes it to a Tcl library procedure for evaluation. The Tcl interpreter parses the command, performs the substitutions described in Figures 2-5, uses the first field of the command to locate a *command procedure* for the command, and then calls the command procedure to actually execute the command. The command procedure carries

out its function and returns a string result, which the Tcl interpreter returns back to the calling code in the application.

The Tcl library includes several *built-in commands* that implement the generic facilities such as variables and looping. Additional command procedures may be provided by each application. The application registers its own specific commands by passing their names and command procedures to Tcl. This information is used later by the Tcl interpreter when it evaluates command strings. Application-specific and built-in commands have exactly the same structure; they are indistinguishable except that built-in commands are registered automatically and users may expect them to be present in all applications. New commands may be created and deleted at any time while an application executes.

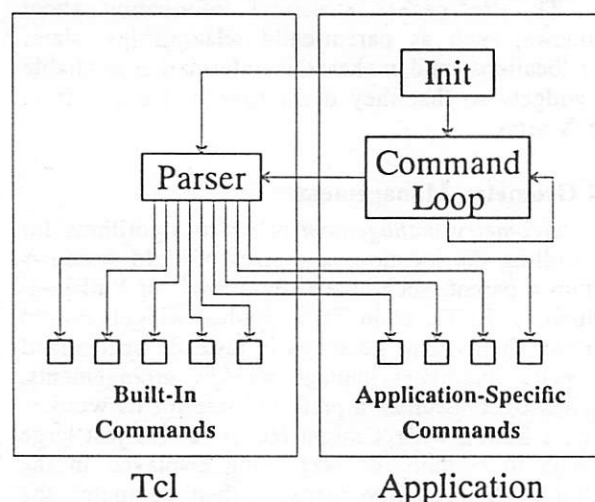


Figure 6: The Tcl interpreter is a C library package that is embedded in applications. The application generates Tcl commands and provides command procedures for application-specific commands. Tcl parses the commands and calls a command procedure to execute each command. Application-specific commands must be registered with the Tcl interpreter, usually during initialization.

Control constructs like `if` are implemented as ordinary commands that make recursive calls to the Tcl interpreter. For example, the command

```
if $i<2 {set j 43}
```

causes the command procedure for `if` to be invoked. This command procedure evaluates its first argument as an expression. If the value of the expression is non-zero, then the command procedure calls the Tcl interpreter recursively to execute the command `"set j 43"`. It is common in Tcl-based applications for one command to take another Tcl command as argument and then execute that command, either immediately or later on.

There is only one official data type in Tcl: strings. All commands, arguments to commands, command results, and variable values are strings. Some commands expect their strings to have particular formats (e.g., arithmetic expressions or Lisp-like lists), but whenever information is passed from one place to another it is as a string. This approach makes it easy to communicate information between C procedures and Tcl programs (there are no complex data type conversions). It also means that Tcl programs have the same basic form as Tcl data, which allows new Tcl programs to be synthesized and executed on-the-fly (in this sense Tcl is similar to Lisp).

The most important aspects of Tcl are the simplicity of the language and the simplicity of its interface to C programs. The language simplicity makes Tcl easy to learn; the interface simplicity makes it easy to use Tcl in applications, easy to write new Tcl commands, and easy to use Tcl to compose primitives written in C.

3. Overview of the Tk Intrinsic

An application based on Tk is constructed by assembling a collection of user-interface components called *widgets*. A widget consists of one or more windows that display information on the screen and react to keystrokes and mouse actions. A widget may be as simple as a "button", which displays a text string and executes a command when a mouse button is pressed over it, or it may be as complicated as a dialog box containing sliders, buttons, text entries, and list boxes. Complex widgets may be composed out of simpler widgets.

As described in the introduction, Tk supports the creation and use of widgets by providing a standard framework in which widgets are constructed; this makes it possible for widgets to be designed and implemented independently yet still work together in interesting ways. Tk also provides a number of convenience procedures to carry out the most common operations required by widget implementors. This set of facilities (the part of the toolkit that isn't associated with a particular widget set) is called the toolkit *intrinsic*.

In Xt and most other toolkits the intrinsic exist as a set of C library procedures. In contrast, Tk provides not only C procedures but also a collection of Tcl commands that make virtually all of the intrinsic accessible from Tcl. The Tcl interfaces allow the look and feel of an application to be queried and modified at any point in the application's execution. They also allow new interface elements, or even new applications, to be created dynamically just by writing Tcl scripts. In these respects Tk is different from other toolkits.

The paragraphs below summarize the main facilities provided by the Tk intrinsics. Most of the facilities are similar to those provided by Xt or other toolkits; where there are differences, they exist mostly to make the Tk facilities accessible from Tcl.

3.1 Window Names

In order to refer to windows in Tcl commands, each window in Tk has a name that identifies it uniquely among all the children of the same parent window. Each window also has a class, such as `Button`, that identifies the type of widget displayed in the window. Lastly, each window has a *path name* that identifies the window uniquely within its application. A path name consists of zero or more names separated by dots. For example, the path name `".a.b.c"` denotes a window `c` inside a window named `b` inside a window named `a` inside the main window of the application. The path name `"."` refers to the main window of the application.

3.2 Event Dispatching

Like most toolkits, Tk provides a centralized mechanism for dispatching X events. Widgets and other interested parties inform Tk of events they care about and provide C procedures to handle the events. When an event occurs Tk invokes all the relevant handlers. The Tk dispatcher supports X events, file events (which trigger when a file becomes readable or writable), timer events, and "when-idle" events (which trigger when all other pending events have been processed).

Tk also provides Tcl commands for creating event bindings; in this case the events trigger the execution of Tcl commands instead of C procedures. See Figure 7 for examples.

3.3 Resource and Structure Caches

Allocating X resources such as pixel values or fonts is expensive because it requires inter-process communication with the X server. To reduce the amount of server traffic, Tk caches information about the X resources currently in use by an application. If the same resource is requested multiple

times for different purposes, only the first request results in server traffic; the later requests are satisfied by sharing the existing resource. This provides a substantial boost in performance in the common case where a few resources are used in many different widgets within an application.

Tk's resource caches are indexed by textual descriptions of the resource rather than binary values (e.g., `MediumSeaGreen` might be used for a color, `coffee_mug` for a cursor, or `@star` for a bitmap stored in a file named `star`). This makes it easier to name X resources in Tcl commands or in the option database described below. In addition, given an X resource identifier, Tk will return the textual name for that resource; this feature makes it easy for widgets to provide human-readable information about their current configuration.

Tk also caches structural information about windows, such as parent-child relationships, sizes, and locations, and makes this information available to widgets so that they don't have to fetch it from the X server.

3.4 Geometry Management

Geometry management refers to algorithms for controlling the locations and sizes of child windows within a parent, such as "all-in-a-row" or "all-in-a-column". In Tk, as in Xt, individual widgets do not control their own geometry. Instead, specialized *geometry managers* manage window arrangements. Each widget specifies a preferred size for its window (e.g., a button widget might request a size just large enough to contain the text being displayed in the button). A geometry manager then computes the actual size for each window, taking into account the requested sizes of the windows it manages, the size of the parent window, and its own particular layout algorithm (see Figure 8 for an example). Each widget must make do with whatever size it is assigned. This approach separates the internal design of a widget from its arrangement in a larger application, so that widgets can be used with a variety of geometry managers.

```
bind .x <Enter> {print "hi\n"}
bind .x a {print "you typed 'a'\n"}
bind .x <Escape>q {print "you typed escape-q\n"}
bind .x <Double-Button-1> {print "mouse at %x %y\n"}
```

Figure 7: Tk provides a Tcl command called `bind`, which can be used to arrange for other Tcl commands to be executed when certain X events (or sequences of X events) occur. The four commands above cause messages to be printed on standard output when the mouse enters window `.x`, when the letter `a` is typed in `.x`, when the escape key is typed followed by the `q` key in window `.x`, or when mouse button 1 is pressed twice in rapid succession in `.x`. Before executing the command for an event Tk replaces `%` sequences in the command with fields from the event. For example, in the last command above the `%x` and `%y` will be replaced with the `x`- and `y`-coordinates from the X event before executing the command.

Tk acts as intermediary for geometry management. It allows geometry managers to claim control over windows, and when a widget requests a particular size for its window Tk passes that information to the relevant geometry manager. Only one geometry manager manages a given window at a time.

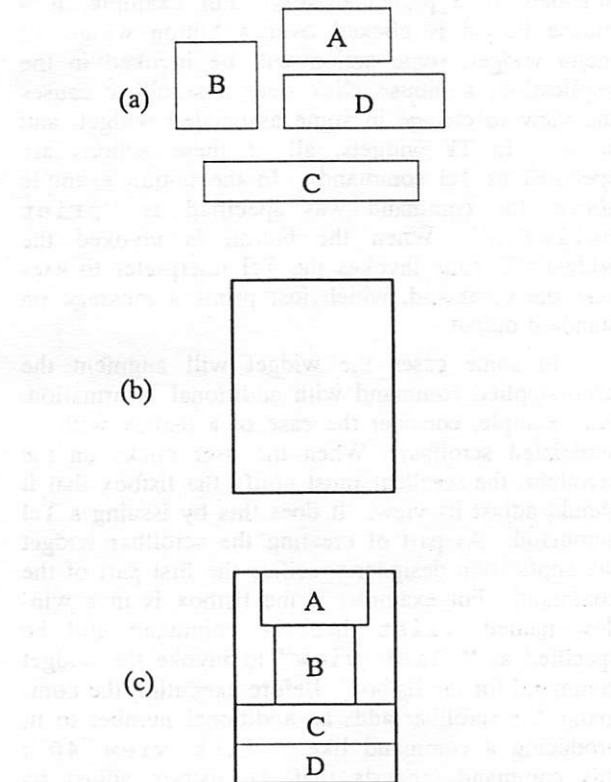


Figure 8: An example of geometry management. (a) shows the requested sizes of four windows and (b) shows the size of the parent window in which the windows are to be arranged. An “all-in-a-column” geometry manager might produce the layout in (c) by arranging the windows in order from top down. Window C ended up with less width than requested and window D received less height than requested because there was insufficient space in the parent. The widgets using windows A-D are expected to make do with whatever size they are assigned by the geometry manager.

Tcl commands are used to control the geometry managers. For example, Tk contains a built-in geometry manager called the *packer*. The command

```
pack append .x .x.a top .x.b top .x.c top
```

will cause the packer to claim control over the windows *.x.a*, *.x.b*, and *.x.c*. The packer will add those windows to the list of windows it manages inside window *.x* and arrange the windows in a column with each window placed at the top of the space not occupied by previous windows in the list. The resulting arrangement will be similar to the one shown in Figure 8. (The packer also includes a

number of other features that are not evident from this one example, such as placing windows against the other sides of the parent’s cavity and selectively stretching windows to fill extra space.)

3.5 Options

Tk provides a standard mechanism for users to specify their preferences about widget options such as colors and fonts. It maintains these options in a database and provides efficient mechanisms for widgets to query the database when they configure themselves. Tk’s option database is the same as the *resource manager* mechanism in Xt: users specify their preferences in a *.Xdefaults* file or in a special root-window property using a simple pattern-matching language (e.g., “*Button.background: red” means that all button widgets should have a red background color). In addition to providing C library procedures for querying the option database, Tk also provides Tcl commands that can be used to query the database or add entries to it.

3.6 The Selection

The X11 Inter-Client Communications Conventions Manual (ICCCM) specifies a complex set of protocols that applications must use to manipulate the selection [10]. Tk provides mechanisms to implement the ICCCM protocols and hide as many of their details as possible. If a widget supports the notion of a selection, it registers a C procedure that Tk may call to retrieve the selection when it is in that widget. This procedure is called a *selection handler* and is similar in many respects to other event-handling procedures. When a widget wishes to claim the selection it calls another Tk procedure, which uses the ICCCM protocols to notify the existing selection owner that it has lost the selection. From this point on Tk will arrange for selection requests to be forwarded to the selection owner by calling its selection handler. When some other widget (potentially in another application) claims the selection, Tk will notify the current owner that it has lost the selection. Lastly, Tk provides a procedure to retrieve the selection from its current owner. Tk also provides Tcl support for the selection: selection handlers may be written in Tcl and a Tcl command is available to retrieve the selection.

3.7 Focus Management

Given that there are many windows on the screen, each of which might potentially receive keyboard input, but only one keyboard, there must be a mechanism for sharing the keyboard among the windows. At any one time keystrokes are directed to a single window, which is called the *focus window* or *input focus*. A separate *window manager* process controls the transfer of the focus among applications,

but the window manager knows nothing of the internal structure of an application. Tk provides a Tcl command that can be used to assign the focus to a particular window within an application, so that all keystrokes in any window of the application are directed to the focus window. For example, when an application pops up a dialog box with a text entry, the focus may be assigned to the text entry so that the user can enter text without having to move the mouse to the dialog box; when the dialog box is complete, it can assign the focus back to the originating window again.

4. Tk Widgets

In Tk, widgets like scrollbars and buttons are implemented with C code that uses the Tk intrinsics. At runtime, Tcl commands are used to instantiate and manipulate widgets. Two different kinds of Tcl commands are used for widgets: creation commands and widget commands. For each type of widget, such as button, radiobutton, or scrollbar, there exists one Tcl command to create widgets of that type. The command's name is the same as the widget's type. For example, the command

```
button .hello -bg Red \
    -text "Hello, world" \
    -command "print Hello!\n"
```

will create a new button widget. The first argument gives the path name of a new window to be created for this widget. Additional arguments are used to specify options for the widget. In the example the options specify a background color to use for the widget, a string to display in the widget, and a Tcl command to execute when the button is invoked by clicking a mouse button over it. For unspecified options, the widget checks in the option database for a value; if none is found then it uses a default associated with the widget type. Once the widget has been created, a geometry manager may be invoked to position the widget in its parent and map the widget so that it is displayed.

As part of creating a widget, a new Tcl command is created whose name is the same as the path name of the widget's window (".hello" in the example above). This command is called a *widget command* and may be used to manipulate the widget. For example, the following Tcl commands could be used to manipulate the button widget created above:

```
.hello flash
.hello configure -bg PalePink1 \
    -relief sunken
```

The first command causes the button to change colors back and forth a few times. The second command resets some of the widget's configuration options: it changes the background color to light pink and changes the 3-D appearance of the button so that it appears to be depressed instead of raised. The `configure` form is supported by all widget

commands and allows any configuration option of any widget to be changed at any time in the same way that it may be specified when creating the widget.

Most widgets are *active*: they carry out some function when manipulated with the mouse and/or keyboard in a particular way. For example, if a mouse button is clicked over a button widget or menu widget, some action will be invoked in the application; a mouse click over a scrollbar causes the view to change in some associated widget, and so on. In Tk widgets, all of these actions are specified as Tcl commands. In the button example above, the command was specified as "`print Hello!\n`". When the button is invoked the widget's C code invokes the Tcl interpreter to execute the command, which just prints a message on standard output.

In some cases the widget will augment the user-supplied command with additional information. For example, consider the case of a listbox with an associated scrollbar. When the user clicks on the scrollbar, the scrollbar must notify the listbox that it should adjust its view. It does this by issuing a Tcl command. As part of creating the scrollbar widget the application designer specifies the first part of the command. For example, if the listbox is in a window named `.list` then the command will be specified as "`.list view`" to invoke the widget command for the listbox. Before executing the command, the scrollbar adds an additional number to it, producing a command like "`.list view 40`"; this command requests that the listbox adjust its view so that item 40 appears at the top of its window.

The use of Tcl commands for all widget actions provides both flexibility and power. In the scrollbar example of the previous paragraph it allowed two independent widgets, a listbox and a scrollbar, to be connected so that they work together. In the most general case a user or application designer could write an arbitrary Tcl procedure and specify that procedure as the command for a widget. In this way, for example, a single scrollbar could be made to control several windows.

5. Programming Within An Application

For many users I expect the Tcl language to be invisible: users will manipulate applications using the keyboard and mouse and be unaware of the fact that an interpretive language underlies the user interface. However, advanced users and application designers can use Tcl to gain power and flexibility. For example, a Tcl command could be invoked to add a new keystroke binding to an existing widget (e.g., backspace over a whole word when Control-w is typed in an entry widget). Such a command could be typed to a running application (if the application provides a command type-in window) or placed in a

startup file to be read automatically whenever the application is executed. The application itself would not have to be modified in any way to support the new binding — as long as the entry widget allows its contents to be fetched and modified from Tcl, it will be possible to implement the backspace-over-word operation using a Tcl command or command procedure.

In addition to all the other purposes it serves, Tcl also serves as a user interface description language; there is no need to design a special user interface language, and Tcl's general programming constructs provide quite a bit of power in creating and modifying interfaces. For example, Tcl can be used to modify the arrangement of windows within an application, e.g., put the diagnostic message window at the top of the application rather than the bottom, or change the order of menus in the pull-down menu bar. Tcl can even be used to create entirely new interface elements such as dialog boxes while an application is running. In fact, Tk contains no special support for dialog boxes. The basic commands for creating and arranging widgets are already sufficient to create dialog boxes: even in the normal case, dialogs are created by writing short Tcl scripts.

Tcl can also be used to create new applications without writing any C code. For example, I have built a simple windowing shell called *wish*, which consists of Tcl, Tk, and a main program that reads Tcl commands from standard input or from a file. Entire windowing applications can be written as scripts for *wish*, just as UNIX commands can be written as scripts for *sh* or *csh*. For example, a simple directory browser can be written as a 21-line *wish* script (see Figures 9 and 10). I plan to enhance *wish* with drawing commands for shapes and text and a few other features; once this is done it will be possible to code a large class of interesting applications entirely in Tcl.

6. Programming Between Applications

In spite of the claims of the previous sections, Tk's greatest benefit of all is not within an application but rather the way it allows different applications to work cooperatively. Currently, the only widely-available communication mechanism between applications is a selection or cut buffer: the user selects information in one application, then invokes a command in another application to retrieve the selection and use it in some way. Besides being limited as a form of communication, this approach is also

```

1 #!wish -f
2 scrollbar .scroll -command ".list view"
3 listbox .list -scroll ".scroll set" -relief raised -geometry 20x20
4 pack append . .scroll {right fill} .list {left expand fill}
5 proc browse {dir file} {
6     if {[string compare $dir "."] != 0} {set file $dir/$file}
7     if [file $file isdirectory] {
8         set cmd [list exec sh -c "browse $file &"]
9         eval $cmd
10    } else {
11        if [file $file isfile] {exec mx $file} else {
12            print "$file isn't a directory or regular file\n"
13        }
14    }
15 }
16 if $argc>0 {set dir [index $argv 0]} else {set dir "."}
17 foreach i [exec ls -a $dir] {
18     .list insert end $i
19 }
20 bind .list <space> {foreach i [selection get] {browse $dir $i}}
21 bind .list <Control-q> {destroy .}

```

Figure 9: A simple directory browser, implemented as a script for *wish*, the windowing shell. This script is stored in a file named *browse* (without the line numbers). Line 1 is a comment line; when the file is executed, it causes *wish* to be invoked as command interpreter for the file. Lines 2-4 create a scrollbar and a listbox, arrange for the scrollbar to control the view in the list box as described in Section 4, and place them side-by-side in the application's main window. Lines 5-15 create a procedure *browse*, which is invoked to browse subdirectories (by running another version of the browser) or files (by running an editor called *mx*). Lines 16-19 initialize the listbox to hold the contents of a particular directory. Lines 20-21 create bindings to invoke the *browse* procedure when space is typed, or to exit when Control-q is typed.

tedious since the user must take some action for each transfer of the selection.

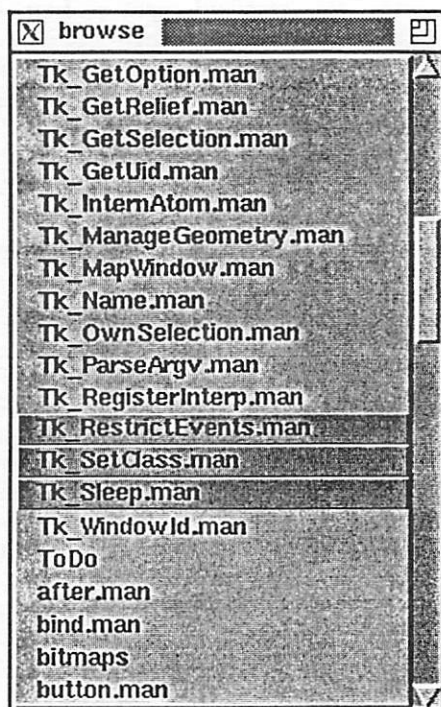


Figure 10: A screen dump showing the appearance of the browser produced by the script in Figure 9. The three darkened items are selected. The window's title bar was generated by the `twm` window manager.

Given such weak communication, application implementors tend to lump functions together into large monolithic applications. This occurs even when the functions are mostly independent, just so that the functions can communicate in ways other than the selection. For example, many debuggers contain built-in editors so that they can display source code and highlight the current line of execution. Commercial spreadsheet programs tend to be lumped together with chart packages, databases, word processors, and communication packages in order to allow the different functions to work together. The lumping results in unnecessary re-implementation of functions: each spreadsheet contains its own chart package, each debugger its own editor, and so on.

Tk solves the problem of poor communication with a Tcl command called `send`. `Send` takes two arguments: the name of an application and a Tcl command. Each Tk-based application has a unique name, and information about all existing applications is registered in a special property on the root window of the display. When `send` is invoked, Tk locates the target application by reading

the registry property. Then Tk forwards the command to the target application (using other window properties). The Tk of the target application executes the command and returns the result of the command back to the originating application. This allows any Tk-based application to control any other Tk-based application on the same display. Any command that could be invoked within an application may be invoked by other applications using `send`, including commands to manipulate the application's interface and also commands to manipulate the application itself.

`Send` is a form of remote procedure call [2]; as such it provides a more general and powerful form of communication than the selection. For example, Tk-based debuggers and editors can be built as separate programs. The debugger can send commands to the editor to highlight the current line of execution, and the editor can send commands to the debugger to print the contents of a selected variable or set a breakpoint at a selected line. A Tk-based spreadsheet might permit cells to contain embedded Tcl commands. When such a cell is evaluated the Tcl command would be executed automatically; it could fetch information from an independent database package or from any other program in the environment. A Tk-based word processor might permit embedded Tcl commands in the body of a document. When the document is formatted, the Tcl commands would be executed; they could retrieve information from spreadsheets, databases, or drawings.

Interface editing provides another example of the power of `send`. Existing interface editors generally operate on application mock-ups. The editor displays something that looks like an application and allows its interface to be edited, but the thing being edited isn't the actual application, so it isn't possible to try out the interface under "real-life" conditions. The interface editor produces an interface description file, which must then be compiled and linked with the application before it can actually be tested. With Tk and `send` it becomes possible for an interface editor to work on live applications, using `send` to query and modify the application's interface. The effects of interface changes can be tested immediately with the application. When a satisfactory interface has been created, the interface editor can produce a Tcl command file for the application to read at startup time to configure its interface in the future.

The overall effect of `send` is that it makes it possible to program applications to work together in powerful ways, so it will no longer be necessary to lump functions into monolithic applications. This encourages the development of lots of small specialized tools that can be programmed with `send` to work together in interesting ways. The tools could be developed and maintained independently, yet be

used in many different ways. I believe that this could result in much richer and more powerful interactive environments than we have today.

The combination of Tcl and Tk and send also allows hypertext and other kinds of active objects to be implemented easily. All that an individual application needs to do is to allow Tcl command strings to be embedded in its internal structures and provide a mechanism for invoking those commands at "interesting" times. Tcl commands can then be written to extend and enhance the behavior of objects. For example, in the spreadsheet envisioned above, commands may be stored in spreadsheet cells; they will be executed whenever the spreadsheet is evaluated. The embedded Tcl commands allow the spreadsheet to "reach out" and retrieve fresh data values from databases or other applications. Or, a hypertext system can be implemented by associating Tcl commands with pieces of text or graphics in an editor; when a mouse button is clicked over an item then the associated commands are executed. A hypertext "link" can be produced by writing a Tcl command that opens a new view and associating that command with some piece of text or graphics. A hypermedia link can be produced using a Tcl command that sends a "play" command to an audio or video application.

7. Status and Measurements

Development of Tcl began in early 1988, and it has been distributed publicly since 1989. The Tcl distribution does not include Tk or any other windowing support. Based on mail I have received about Tcl, I estimate that about 50 Tcl-based applications exist or are under construction.

I began implementing Tk in late 1989. At present the intrinsics are complete, although they are evolving rapidly as I gain experience using them to

implement widgets. I have built a number of Motif-compatible widgets, including panes, labels, buttons, check buttons, radio buttons, messages, list-boxes, scrollbars, and scales. Two major widget types, entries and menus, are still left to be implemented (I hope to complete both of these before this paper is published). I expect to begin distributing Tk in early 1991. As with Tcl, the code will be freely distributed without any licensing restrictions.

Table 1 shows the sizes of Tk and Tcl in lines of code and in compiled bytes, and compares them to the sizes of corresponding portions of the Xt toolkit and the Motif widget set. Tk and Tcl together have only three-quarters the compiled size of Xt, even though they provide more flexibility and power. Tk's widgets and geometry manager are 2-5x smaller than the corresponding Motif modules. As Tk's widgets mature I expect them to grow slightly, but I believe that their final sizes will still be substantially smaller than the Motif widgets.

Tcl simplified Tk and its widgets by making a single unifying language available everywhere in the system. Tk implements only a few key primitives, which can then be composed with Tcl. In systems without a composition language, such as Xt/Motif, all run-time needs must be predicted and addressed explicitly in the C code; this increases the amount of code that must be written. In addition, the lack of a single unifying language resulted in many different protocols and "little languages" to handle different situations in Xt and Motif (examples are the ICCCM selection protocols, the Xt translation manager, and Motif's UIL interface description language). These additional protocols add to the complexity of the system.

Table 2 gives a few sample performance numbers for the Tk toolkit. On a machine with 10 MIPS or more, the Tcl interpreter is fast enough to

	Source Lines		DS3100 bytes	
	Xt/Motif	Tk	Xt/Motif	Tk
Intrinsics	24900	15100	216400	92800
Tcl		9300		61100
Geometry Manager	2100	1000	17100	7400
Buttons	6300	1000	43700	8600
Scrollbar	3000	1200	24900	8000
Listbox	6400	1600	53100	10700
Total	42700	29200	355200	188600

Table 1: A comparison between Tk and Xt/Motif based on lines of source code and bytes of compiled object code (for the DECstation 3100) for selected modules. "Geometry Manager" refers to the PanedW module in Motif and the "packer" geometry manager in Tk; the packer is somewhat more general and flexible than PanedW. "Buttons" consists of three files in Motif (Label, PushB, and Toggle); in Tk a single file implements labels, buttons, check buttons, and radio buttons. The totals reflect only the modules in the tables; Both Tk and Motif contain additional widgets not reflected in the table.

execute many hundreds of Tcl commands within a human response time; this permits relatively lengthy Tcl scripts to be executed without noticeable delays. The `send` command currently takes a few tens of milliseconds. At this speed, it is possible to paint with the mouse in one application, have all the mouse motion events bound into Tcl commands, which in turn use `send` to forward commands to another application in a different process, which finally draws the painted object in its own window, and have all of this take place with no noticeable time lag. Tk is fast enough to instantiate relatively complex applications (many tens of widgets) in a fraction of a second. Tk has not undergone any performance tuning yet; when it does there should be some improvement in these numbers.

Operation	Time
Simple Tcl command (<code>set a 1</code>)	68 μ s
Send empty command	15 ms
Create, display, delete 50 buttons	440 ms

Table 2: Execution times for selected operations in Tk. All times were measured on a DECstation 3100 running Ultrix 4.2 and X11R4. In the bottom measurement of the table, about half of the elapsed time was spent executing in the client and about half in the X server.

8. Comparisons

Of the existing X11 toolkits, Tk is most similar to Xt [1]. The major facilities provided by Tk were inspired by Xt and are similar to the corresponding facilities of Xt. There are also similarities between Tk and the InterViews and Andrew toolkits [5,9] in that all support some sort of widget-like notion to decompose applications. However, InterViews and Andrew have more support for the underlying application object structures whereas both Tk and Xt focus almost exclusively on the interface aspects, with little support for the application structures.

The most significant difference between Tk and the other toolkits is the presence of Tcl in Tk. Run-time languages are starting to appear in other systems, such as Ness, which is used to embed executable programs into documents in the Andrew toolkit [3], and UIL, which is used to specify interfaces in Motif [7]. However, these languages have three disadvantages relative to Tcl. First, they are less dynamic. For example, UIL programs must be compiled before being processed by a running application, and Ness appears to require many decisions to be made statically. In contrast, Tcl is interpretive, so any available operation can be invoked at any time. Second, the other languages are less complete. For example, UIL does not include control constructs such as `if` and `while`, and Ness functions are not first-class objects. In contrast, Tcl is a

complete programming language that even provides access to its own internals (e.g., it is possible to retrieve the body of a Tcl procedure or a list of all defined variable names). Third, the other languages are special-purpose: they only control a portion of an application's functions. In contrast, Tcl is used for virtually all aspects of an application, which makes it possible to compose all of those aspects to work together.

Another difference between Tcl and other toolkits is the `send` command for inter-application communication. I know of no equivalent construct in other X toolkits. The closest existing facility is Microsoft Windows' Dynamic Data Exchange protocol (DDE), which allows applications to communicate in several ways including passing commands for remote execution [6]. However, for remote execution to be most useful it must allow access to all the internals of the remote application. For this to happen, the language used by the remote execution facility should be the same as the language used to control the user interface and internals of the target application, as it is with Tcl and Tk. Unfortunately, the Windows environment does not include a universal command language. Although a standard syntax is suggested for remote commands, there is no built-in connection between these remote commands and the internals of the remote application. Each application must provide special code to parse and execute all the remote commands it wishes to support. This will probably limit the use of remote execution in DDE to a small set of functions. In contrast, Tk's `send` command provides access to all aspects of other Tk-based applications without any extra effort on the part of the applications' developers.

One final difference between Tk and other toolkits is object orientation. InterViews, Xt, and Andrew are all strongly object-oriented with support for classes and inheritance. In contrast, Tk is not strongly object-oriented. The widget commands described in Section 4 give Tk an object-like feel, and Tk makes extensive use of procedure variables and callbacks, but there is no official class mechanism and no inheritance among widget types. Instead of providing inheritance, Tk focuses on *composition*: mechanisms for assembling independent widgets into interesting arrangements. In my opinion, composition is more important for a toolkit than inheritance. There isn't enough commonality between widgets for inheritance to provide much benefit, and inheritance adds complexity (to understand one widget you must understand all the widgets it inherits from). Inheritance mechanisms only benefit a small group of people (widget implementors), whereas composition mechanisms allow any user to create new interface elements out of existing widgets. Further support for this view comes from the InterViews system: although it is written in C++ and claims to be

object-oriented, the primary benefit claimed for the system is its support for composition [5].

9. Conclusions

I believe that Tk provides a large increase in power and flexibility over existing windowing toolkits. Tk's power comes from two sources: the power of programming and the power of building interchangeable tools. The use of Tcl within Tk (and within Tk-based applications) means that a single programming language is available at run-time to control all aspects of an interactive application, from its look to its feel to its function. This in turn makes it possible to modify and extend all of these aspects of an application at any time. The second source of power is from composition: the ability to build independent units that can work together and be re-used in many different ways unforeseen by their designers. Tcl acts as a composition language both for composing widgets within an application and for making different applications work together.

I hope that Tcl and Tk can do for interactive applications of the 1990's what the UNIX shells did for stream-based applications of the 1970's. The UNIX shells encouraged the construction of small tools that read from standard input, perform some operation on the data, and write the results to standard output. The shells provided mechanisms for these "filters" to be hooked together in many different ways to perform interesting functions. I hope that Tcl and Tk will encourage the development of many small specialized windowing tools that present simple Tcl interfaces. Tk permits the tools to work together by sending commands to each other. With this approach I hope it will become possible to build more powerful interactive applications with much less effort than is needed today.

10. Acknowledgments

The work described here was supported in part by NASA and the Defense Advanced Research Projects Agency under Contract No. NAG2-591, and in part by Digital Equipment Corporation. Joel Bartlett, Mike Kupfer, Joel McCormack, and Ken Shirriff provided numerous suggestions that improved the presentation of the paper. The design of Tk benefited from discussions with Joel Bartlett and Joel McCormack; I am grateful for their stimulating comments and criticisms.

11. References

- [1] Asente, P. and Swick, R., with McCormack, J. *X Window System Toolkit: The Complete Programmer's Guide and Specification*. Digital Press, 1990.
- [2] Birrell, A. and Nelson, B. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1986, pp. 39-59.
- [3] Hansen, W. "Enhancing Documents With Embedded Programs: How Ness Extends Insets in the Andrew Toolkit." *Froc. 1990 International Conference on Computer Languages*, March 1990.
- [4] Libes, D. "expect: Curing Those Uncontrollable Fits of Interaction." *Froc. USENIX Summer Conference*, June 1990, pp. 11-15.
- [5] Linton, M., Vlissides, J., and Calder, P. "Composing User Interfaces with InterViews." *IEEE Computer*, Vol. 22, No. 2, February 1989, pp. 8-22.
- [6] *Microsoft Windows Software Development Kit, Guide To Programming, Version 3.0*. Microsoft Corporation, 1990.
- [7] *OSF/Motif Programmer's Guide, Revision 1.0*. Prentice Hall, Englewood Cliffs, NJ, 1990.
- [8] Ousterhout, J. "Tcl: An Embeddable Command Language." *Froc. USENIX Winter Conference*, January 1990, pp. 133-146.
- [9] Palay, A., et al. "The Andrew Toolkit - An Overview." *Froc. USENIX Winter Conference*, February 1988, pp. 9-21.
- [10] Scheifler, R., and Gettys, J., with Flowers, J., Newman, R., and Rosenthal, D. *X Window System: The Complete Guide to Xlib, X Protocol, ICCCM, XLFD* (Second Edition). Digital Press, 1990.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California at Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. In addition to his work on command languages and toolkits, he is currently leading the development of Sprite, a network operating system for high-performance workstations. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980. His electronic mail address is ouster@sprite.berkeley.edu.



User Interface Construction Based On Parallel and Sequential Execution Specification

Toshiyuki Masui – Center for Machine Translation, Carnegie Mellon University

ABSTRACT

The user interface part of an application program can be easily and compactly constructed by combining the parallel execution primitive *Linda* and the state transition description language *Flex* with a general purpose programming language. With this approach, a wide range of interfaces can be constructed without using U/I-specific languages or systems. Using these tools, parallel execution, separation/communication between the application and the interface part, and complicated dialogs can easily be specified. In our implementation, the specification is compiled into C++ and runs efficiently without any runtime system.

Introduction

Many kinds of interaction techniques are now available on UNIX workstations. Various tools have been developed to help building complex interfaces easily. There are two major tools for interface specification. One is the which is a set of interface library functions. The other is the (User Interface Management System)[1, 2] which urges the separation between the application and the interface part.

Although those tools are useful in many cases, both of these tools have intrinsic disadvantages. First, we show the indispensable features required for an interface building tool. Second, we point out the problems of existing interface building tools. Third, we show that these problems can be solved by the combination of a parallel execution primitive and a state transition specification tool with a general purpose language. Finally, we discuss the problems of these tools.

Essential Features for an Interface Building Tool

We list here the essential features required for a user interface construction tool.

Separation between Application and Interface Part: The interface of a program can be replaced easily if the interface portion is separated from the application portion. This feature is also useful for rapid prototyping of the interface, for interfaces can be developed separately even when the application part is not completed yet. Separation brings many advantages.

Cooperation with Application Program: Although separating the application part from the interface part has many advantages, strict separation introduces other problems, since the application and the interface must communicate with each other in some way to achieve semantic

feedback.

Parallel Execution Specification: The ability to handle multiple input and output devices is often desirable for a good interface. If we use a process for each device, the structure of the interface program becomes much simpler than if we handle all devices from one process. Also, if we can create separate processes for the application, interface, and other (e.g. testing) parts of the program, developing and maintaining each part becomes much easier than developing an all-in-one system. Thus, parallel execution is essential for the specification of user interfaces.

Dialog Specification: A good specification technique is needed to specify a complicated dialog with many modes, conditions and exceptions. Since human input is neither consistent nor based on a simple grammar, all kinds of special cases must be considered. As it is difficult to specify a complicated dialog a good tool for interface specification should be used.

Compactness: Interface programs must be implemented compactly on actual machines. Implementations which can run only on large machines or require a lot of resources would not be used in embedded systems or portable computers. Interfaces must be implemented compactly for such environments.

Problems of Existing Tools

Problems of User Interface Toolkits

Currently, many types of toolkit on various window systems have been proposed and sold as products. It is quite easy to create a pretty-looking interface gadget by using a toolkit. It is also very convenient to be able to create a decorated window with scrolling bars or a text input window with a

cursor just by calling a single function. A toolkit is also useful because it standardises the interface. However, toolkits have several disadvantages.

First, *the interface cannot be specified separately from the application*. The interface portion is usually called from the application portion via a function call and the dialog cannot be specified separately. Generally speaking, using a toolkit is not helpful for the separation of application and interface. Second, *the structure of the application is determined by the functionality of the toolkits*. Toolkit functions must be called in a predetermined order and the application program must be written to keep that order. The structure of an application program depends entirely on the structure of the toolkit environment, which includes the kind of operating system, the window system, the language, etc. Third, *interface programs are limited to those provided by the toolkits*. Unsupported patterns of interface cannot be implemented. For this reason, toolkits tend to provide a lot of functions to respond to the demands of various kinds of applications, and their libraries become huge. It is usually quite hard for an application programmer to know the whole capability of a toolkit.

Problems with UIMS

Although UIMS solve some problems of toolkits, they also have other problems of their own [2].

As the primary aim of a UIMS is the separation between application and interface, the interface part can be described separately from the application part. Special dialog description languages are often used to specify the interface. However, as an interface description often requires various calculations, the interface description language has to provide features found in general programming languages and tends to become complicated. In such a case, users may prefer writing interface programs by himself without any help of UIMS. Moreover, UIMS are usually big systems, and applications using UIMS may run slower than ones which do not use UIMS. Programs for small machines and programs requiring high performance therefore cannot use such a UIMS. For these reasons, UIMS are not yet in popular use.

Basis for Interface Construction

We now describe the basis for an interface tool by considering the indispensable features.

Parallel execution primitive: As we mentioned before, a parallel execution description is essential for interface specification. Separation and cooperation between application and interface can be achieved by a good parallel execution primitive.

Dialog specification tool: Some method must be used to specify a dialog which contains

complicated state transitions. As general programming languages are not suitable for the description of state transitions, we need a good dialog description tool.

Implementation in a general programming language: Although it is desirable to write the interface part of a program in the same programming language as the application part, the former requests features usually not provided in conventional programming languages. But as we discussed before, This should not be achieved by using a special interface language. We thus adopt the strategy of *adding flavors of parallel execution and state transition description* to a general programming language.

With the considerations outlined above, the basis of user interfaces description can be described as follows:

**UI Description =
Parallel Execution Primitive +
State Transition Specification
in a General Programming Language**

With this approach, many kinds of interface can be created easily with other existing techniques.

Parallel Execution Primitive

Many parallel execution primitives have been proposed to describe interfaces. Examples are CSP [3], coroutines [4], Switchboard [5], and ERL [6]. These are all special languages and they cannot be used with conventional programming languages.

We propose the use of the parallel execution primitive Linda [7] for the description of interface interaction.

Linda is a space- and time-uncoupled parallel description language. Processes communicate with each other using tuples (sets of data) in the (or TS). Unlike former parallel description languages, Linda is novel for its simplicity and expressive power.

A Description of Linda¹

Linda provides four basic operations: `eval` and `out` to create new data objects; `in` and `rd`, to remove and to read them respectively.

The sender process creates a new tuple via `out`. The receiver process deletes a tuple from tuple space via `in`.

A tuple, unlike a message, is a data object in its own right. In message-sending systems, a message must be directed to some receiver explicitly, and only that receiver can read it. Using Linda, any number of processes can read a message; the sender need not know or care how many processes or which

¹This section is based on [8].

ones will read the message. Processes use the `rd` operation to read a tuple without removing it.

A new process is created by `eval`.

The fact that senders in Linda need not know anything about receivers and vice versa is central to the language. It promotes the so-called uncoupled programming style. When a Linda process generates a new result that other processes will need, it simply dumps the new data into tuple space. A Linda process that need data looks for it in tuple space.

A tuple exists independently of the process that created it, and in fact many tuples may exist independently of many creators, and may collectively form a data structure in tuple space. Tuples are referenced associatively, in many ways like the tuples in a relational database. A tuple is a series of typed fields, for example

```
("a string", 15.01, 17,
    "another string")
```

and `(0,1)`.

Executing the `out` statements:

```
out("a string", 15.01, 17,
    "another string")
out(0,1)
```

causes these tuples to be generated and added to tuple space.

`out` statements do not block. A process executing `out` continues immediately.

An `in` or `rd` statement specifies a template for matching. Any values included in the `in` or `rd` must be matched exactly; formal parameters must be matched by values of the same type.

Consider this statement:

```
in("a string", ? f, ? i,
    "another string")
```

Executing this statement causes a search of tuple space for tuples of four elements, where the first element matches "a string", the last element matches "another string", and the middle two elements are of the same type as variable `f` and `i`. When a matching tuple is found it is removed from tuple space, the value of its second field is assigned to `f`, and the value of the third field is assigned to `i`. If there are no matching tuples when `in` executes, the `in` statement blocks until a matching tuple appears. If there are many, one is chosen nondeterministically.

The statement

```
rd("a string", ? f, ? i,
    "another string")
```

works in the same way, except that the matched tuple is not removed. The values of the middle two fields are assigned to `f` and `i` as before, but the tuple remains in tuple space.

Advantages of Linda for Interface Specification

Linda is suitable for describing user interfaces because of the following factors:

Simplicity: There are only four operations: `out()`, `in()`, `rd()`, and `eval()`. The meanings of these operations are very simple.

Time and Space Uncoupling: As interprocess communication in Linda is time- and space-uncoupled, application and interface can work independently. They can even work without knowing each other. They must only know the definition on TS. This feature is useful for the separation of application and interface. Actually, there is no need to distinguish application and interface.

Communicating through tuple space has other advantages. Consider a help facility. We want to use the help facility with the same interface whenever possible. But adding a condition for help facility to each dialog description is quite tedious and error-prone. With Linda, once we define a process which implements the help operations, we can utilize the help facility anytime by simply putting a tuple for the help operation into TS. Also, testing the interface program is easily accomplished by just exchanging the input process with a testing process.

Moreover, separate modules for managing various tasks can be implemented as separate processes. For example, a constraint manager which takes care of the arrangement of the display, and a time manager which takes care of time-outs can run as separate Linda processes.

Language Independence: Most high-level parallel description primitives are languages by themselves. Linda can be embedded in general programming languages as a library extension.

Flexibility of Event Handling: Since every process can put/get events to/from TS, every process can work as an event dispatcher. Also, since simple pattern matching method is used in `in()` and `rd()`, flexible event handling is possible. For example, while one application is waiting for any input event by `in('event', ? type, ? arg)`, another can wait for only keyboard events by `in('event', 'key', ? arg)`.

Efficiency of Implementation: Linda can be efficiently implemented on single processor machines.

Communications via TS is shown in Figure 1. Each white oval denotes a process executing a Linda operation. The process which executes

```
in('event', 'text', ? s)
```

can get the tuple

```
('event', 'text', "abc")
```

regardless of whether it was put into TS by a keyboard input process or a tablet input process.

Applications which want to get a string event need not know where it was from. They can also get the event selectively simply by using a more specialized in. Time and space independence and selectivity of Linda is essential for interface description.

Implementation

In our system, Linda is currently implemented in C++. However, Linda can be used with any language, and similar arguments apply to other languages.

State Transition Specification

Need for State Transition Specification

It is well known that state transition diagrams are useful for dialog specification. Some authors have proposed user interface specification tools which only utilize state transition diagrams [9, 4, 10]. For example, Jacob [4] proposed a UIMS which combines state transition descriptions and coroutines. All of them use some special language for interface description.

The need for a state transition specification tool is not limited to user interface design. For example, let us consider an "Escape Sequence Interpreter" for a display terminal emulator². With a conventional

²Many intelligent terminals can move the cursor, reverse characters, etc., when they receive a string which starts with an ESC(0x1b) character. Those sets of special strings differ from terminal to terminal. Those strings are called "escape sequences."

programming language like C, a terminal could be programmed as follows.

```
c = getchar();
switch(c){
  case ESC: {
    c = getchar();
    switch(c){
      case '[': ..... break;
      default: ..... break;
    }
  }
  default:
}
```

The execution states are not explicitly described, and the location of each statement stands for the state of this program. The whole program structure may have to be modified even when only a small part of the state transition has changed. This kind of program is error-prone and hard to maintain. State transitions should be specified separately.

Flex: A State Transition Specification Tool

State transitions can be specified by a regular expression which denotes a finite state automaton. is a tool developed by our group which converts the specification of state transitions into a C program which interprets it. The specification of Flex is like that of *lex*(1). Unlike *lex*, which invokes the specified action when the longest matching sequence is found, Flex invokes the specified action as soon as the specified pattern is matched. Flex can also create many state transition machine objects which

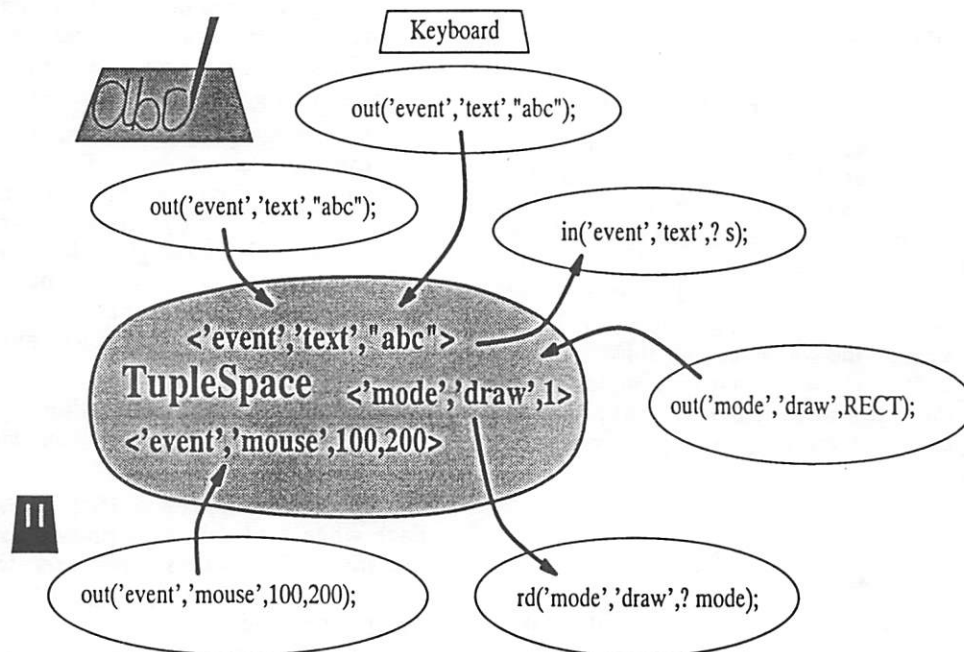


Figure 1: Communication between Application and Interface via TS

can accept the same sequence.

Grammar of Flex

A Flex source consists of 3 parts separated by `%%`.

```
<head definition part>
%%
<statename><pattern> <action defn>
%%
<tail definition part>
```

Definition parts are copied to the output program. A *statename* is used to specify the state where the *pattern* is expected. If it is omitted, the initial state is used instead. The *pattern* and *action definition* part are converted by Flex to a state transition machine which accepts the input that matches the specified regular expression. The specified action is invoked when a match occurs. The action definition part consists of statements enclosed by a pair of braces. A BEGIN statement in the action definition part designates an explicit state transition.

Example Usages of Flex

Escape Sequence Interpreter

The escape sequence interpreter shown before can be programmed with Flex as shown below:

```
%%
\033\{ /* action when 'ESC [' is accepted */ }
[^\033] { /* characters other than ESC */ }
%%
```

A state transition table and function is created from the specification.

```
%%
a      { print("あ"); }
ba     { print("ば"); }
bb     { print("っ"); pushback(); }
be     { print("へ"); }
bi     { print("ひ"); }
bo     { print("ほ"); }
bu     { print("ぶ"); }
bya    { print("ひゃ"); }
bye    { print("ひょ"); }
byi    { print("ひい"); }
byo    { print("ひょ"); }
byu    { print("ひゅ"); }
cc     { print("っ"); pushback(); }
cha    { print("ちゃ"); }
%%
```

Example 1: ASCII to Kana conversion

ASCII to Kana Conversion

Here is another example where Flex is useful. Converting an ASCII character sequence into a Kana (simple Japanese character sets) is necessary for the input of Japanese text from an ASCII keyboard.

Each Kana character corresponds to 1 to 3 ASCII characters. For example, the two-character sequence "ba" is converted to the Kana character "ば". The ASCII to Kana conversion program can be specified by Flex as shown in Example 1. The function `print()` prints its argument string; the function `pushback()` returns the last character to the input for reuse. This kind of program cannot be written in conventional programming languages without using many conditional branches or ad hoc method like a consonant/vowel table. These programs are very difficult to debug.

Range of Flex Specification

Transition to another state is specified by a BEGIN statement in the action definition part. BEGIN statements can be used with arbitrary statements like conditional branches, loops, etc. With this feature, the language accepted by Flex is not limited to regular languages. For example, in the program shown below, 10 'b's after one 'a' will cause a state transition to `S3`.

```
int count;
%%
a      { count = 10; }
b      { if(--count == 0) BEGIN S; }
<S>
%%
```

An Example of Dialog Specification with Flex

An example usage of Flex for describing the dialog of Kana-Kanji conversion is shown below. Kanji is a complex character set used for Japanese/Chinese texts. As there is no straightforward way to specify a Kanji character from an ASCII keyboard, two-step method is usually taken. First, ASCII string is converted to Kana string by the method described before. Next, the Kana string is converted to Kanji string of the same pronunciation. The second conversion is a hard task, for there are many Kanji characters of same pronunciation. By way of example, "漢字", "幹事", and "感じ" have the same pronunciation as "kanji."

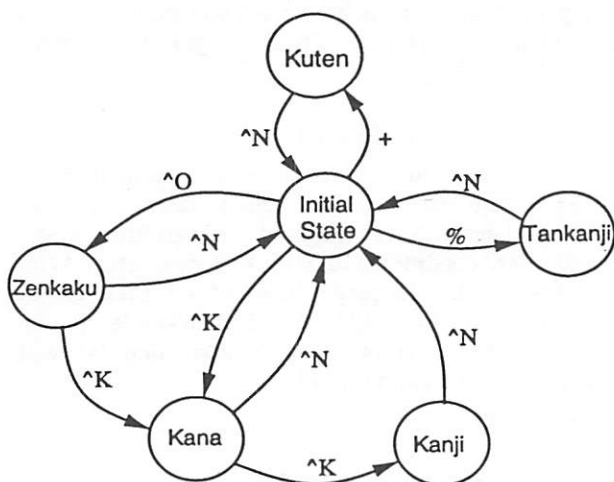
There are many methods of Kana-Kanji conversion and a very simple one is used here. In the initial state, ASCII characters can be input. When Ctrl-K is pressed, the ASCII string is converted to Kana string. When Ctrl-K is pressed again, it is converted to Kanji string. When Ctrl-O is pressed while no conversion has been made, the ASCII string is converted to Zenkaku string, which corresponds to the same ASCII string with large fonts⁴. At each state, the conversion resumes to the

³The same discussion applies to lex.

⁴Kanji characters are usually displayed twice as large as ASCII characters. Zenkaku ASCII characters are displayed in the same size as Kanji characters.

initial state when Ctrl-N is pressed.

Below is the state transition diagram of this interface and the specification in Flex. Tankanji conversion (simple conversion method where an ASCII character string is directly converted to a Kanji character) and Kuten conversion (code number is converted to a Kanji character) is also available here.



```

%%
\x0f      {... BEGIN Zenkaku;}
\x0b      {... BEGIN Kana;}
%         {... BEGIN Tankanji;}
+         {... BEGIN Kuten;}
<Zenkaku>\x0e {... BEGIN 0;}
<Zenkaku>\x0b {... BEGIN Kanji;}
<Kana>\x0e   {... BEGIN 0;}
<Kana>\x0b   {... BEGIN Kanji;}
<Kanji>\x0b  {... }
<Kanji>[^\x0b] {... BEGIN 0;}
<Tankanji>.  {... BEGIN 0;}
<Kuten>.     {... BEGIN 0;}
%%

```

Figure 2: State transition diagram of Kana-Kanji conversion and its specification in Flex

Although the conversion method is complicated and has several states, the specification in Flex is quite simple and the specification is converted to an efficient C++ code. Since ASCII-Kana and Kana-Kanji conversion are hard dialog examples, it is shown that Flex is powerful enough for the specification of any complicated dialog.

Examples of Interface Specification

We will show some examples which make clear that complicated interfaces can be easily constructed through the combination of Linda and Flex. We first show that a toolkit can be constructed easily with these tools. We next show an example bitmap editor built on the toolkit where concurrent input

from multiple devices is available. We last show an FSM editor where displayed objects are constrained by each other's location.

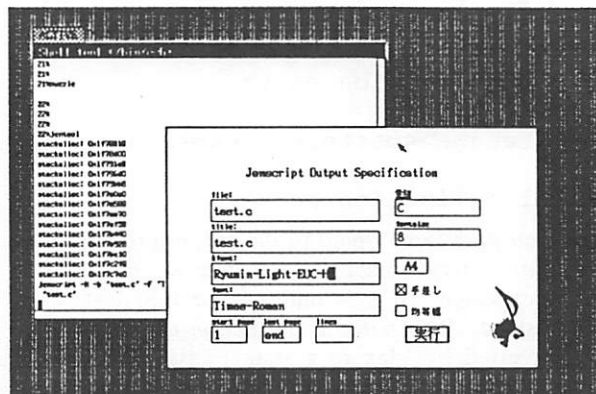


Figure 3: Print Format Specification Program

A Toolkit Based on Linda and Flex

A Toolkit can be easily constructed using Linda and Flex. Figure 3 is an example of a print format specification program which uses this sample toolkit.

Kanji strings can be input to the frame (text box) with a cursor. Kana-Kanji conversion can be done independently to other text boxes. A check mark displayed near “手差し” (manual feed) is toggled by each click in that box and indicates the selection of that item.

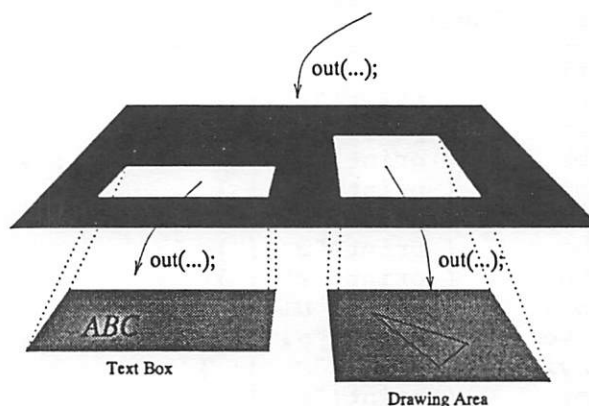


Figure 4: Hierarchy of the Toolkit

Structure of the Toolkit

In Figure 3, each frame corresponds to a part of the UI toolkit, and processes are attached to each frame. Portions of the tool are arranged as a hierarchy corresponding to their appearance. Tools at a higher level send tuples to lower-level tools when necessary. The whole window corresponds to the root (highest) tool.

In Figure 4, each rectangle corresponds to a tool object. If the higher-level (darker) tool receives a tuple and finds that the tuple is for the lower-level (lighter) tool, it sends the tuple to the lower-level tool. In Figure 3, the "Jenscript Output Specification" window corresponds to the higher-level tool and other small rectangles correspond to the lower-level tools.

Basic Operation of Each Tool

Each process is created by `eval()` and continues executing the following loop.

```
for(;;){
  in(ToolId, ? EventName, ? arg);
  <actions corresponding to the tool>
}
```

In other words, each tool is waiting for a tuple.

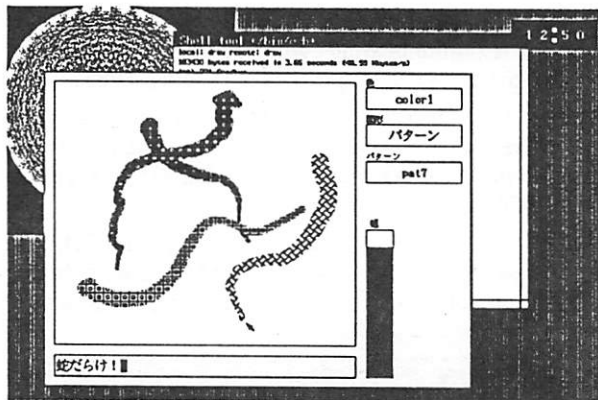


Figure 5: Bitmap Editor

A Simple Tool

A simple tool like checkbox can be programmed as follows:

```
for(;;){
  int value;
  in(ToolId, ? Event Name, ? arg);
  in(ToolId, ? value)
  <toggle the value>
  out(ToolId, value)
  <draw or erase the check mark>
}
```

The state transition is not explicitly specified here, for there are only two states (value on and off). As the tuple `(ToolId, value)` is stored in the global tuple space, any process can read the current value of value via `rd(ToolId, ? value)`.

A Complex Tool

A textbox contains many states, each of which corresponds to a state of Kana-Kanji conversion. Although state transition of Kana-Kanji conversion is complicated, as shown in Figure 2, it can be

specified simply by Flex. The complexity of state transition is hidden from other tools, and does not add to the complexity of other tools. Changes in textbox do not affect other tools.

An Example of Parallel Execution

A color bitmap editor using this toolkit is shown in Figure 5.

The slide-bar at the right side with the Kana title "幅" (width) is a tool for specifying a value. The value can be changed not only by moving the mouse over the slide bar, but also by keyboard input. In this example, the value corresponds to the width of the line drawn by the mouse in the main drawing area. In Figure 5, "snakes" are drawn by drawing contiguous circles by dragging the mouse while changing the width via the keyboard. This can only be done by the parallel execution of each tool [6].

Evaluation

As we have shown above, a toolkit with parallel execution primitives is easily constructed through the combination of Linda and Flex. In this toolkit, each tool is working as an independent process and decides which event to use and which event to pass to other processes. The toolkit is the collection of simple tools which do not need to know what other tools do. In existing toolkits, one central event manager first gets all the events and it dispatches them to each tool. Each tool depends on the behavior of the event manager. Usually, tools and event manager cannot run in parallel. Although there exist some UIMS and toolkits which can handle parallel execution of application and interface, they all use special languages and specification techniques. It is worth noticing that the combination of simple and general tools like Linda and Flex is powerful enough for interface specification.

Implementation of Constraint Programming

Constraints have long been used for managing graphic objects [11, 12]. Recently, many people have advocated constraints as a suitable method to represent the relation between graphic objects in a graphical UIMS [13, 14, 15, 16]. We now show that constraint management of display objects can easily be handled by Linda.

Figure 6 shows a state transition editor which utilizes constraints between display objects. Each arc is constrained by the position of two circles on both sides (which indicates states) and the position of a point on the arc. Each arc is redisplayed when the position of a circle or a point is changed. Each circle and arc corresponds to a process which is waiting for the change of the constraint by `in()`. The process also notifies the change to other objects by `out()` when needed. State transition diagrams like the one shown in Figure 2 can easily be written by this editor.

Discussion

Although this approach works well for many applications, there are several problems. First, as parallel process execution contains some overhead, a program using Linda either runs slower or requires larger space than a program which uses no parallel execution primitive. Second, the specification in Flex is sometimes at too primitive a level to express a high-level interface. Finally, as the specification must be compiled, interactive creation of interface is impossible.

The first problem could be solved if we could devise an efficient implementation of Linda. A Linda implementation for multiple processors would work well for an embedded system with several microprocessors.

The second problem should be solved by creating some other high-level interface primitives based on Linda and Flex.

As the last problem conflicts with the need for compact compilation, a run-time interface development environment should be created.

Conclusions

The combination of the parallel execution primitive Linda and the state transition specification tool Flex is shown to be a powerful basis for constructing user interfaces. They can be implemented efficiently

and used with arbitrary general purpose programming languages. Interface specification techniques such as constraint programming can be used in combination with these tools.

Acknowledgements

We thank Prof. Dario Giuse for numerous comments and advice for this paper. We also thank Prof. Masaru Tomita for various supports for the research.

References

1. J. Lowgren, "History, State and Future of User Interface Management Systems," *SIGCHI Bulletin*, pp. 32-44 (July 1988).
2. B. A. Myers, "User-Interface Tools Introduction and Survey," *IEEE Software* 6 pp. 15-23 (January 1989).
3. L. Cardelli and R. Pike, "Squeak a Language for Communicating with Mice," *Proceedings of SIGGRAPH 19* pp. 199-204 (July 1985).
4. R. J. K. Jacob, "A Specification Language for Direct-Manipulation Interfaces," *ACM Transactions on Graphics*, pp. 283-317 (October 1986).
5. P. P. Tanner, S. A. MacKay, and D. A. Stewart, "A Multitasking Switchboard Approach to User Interface Management,"

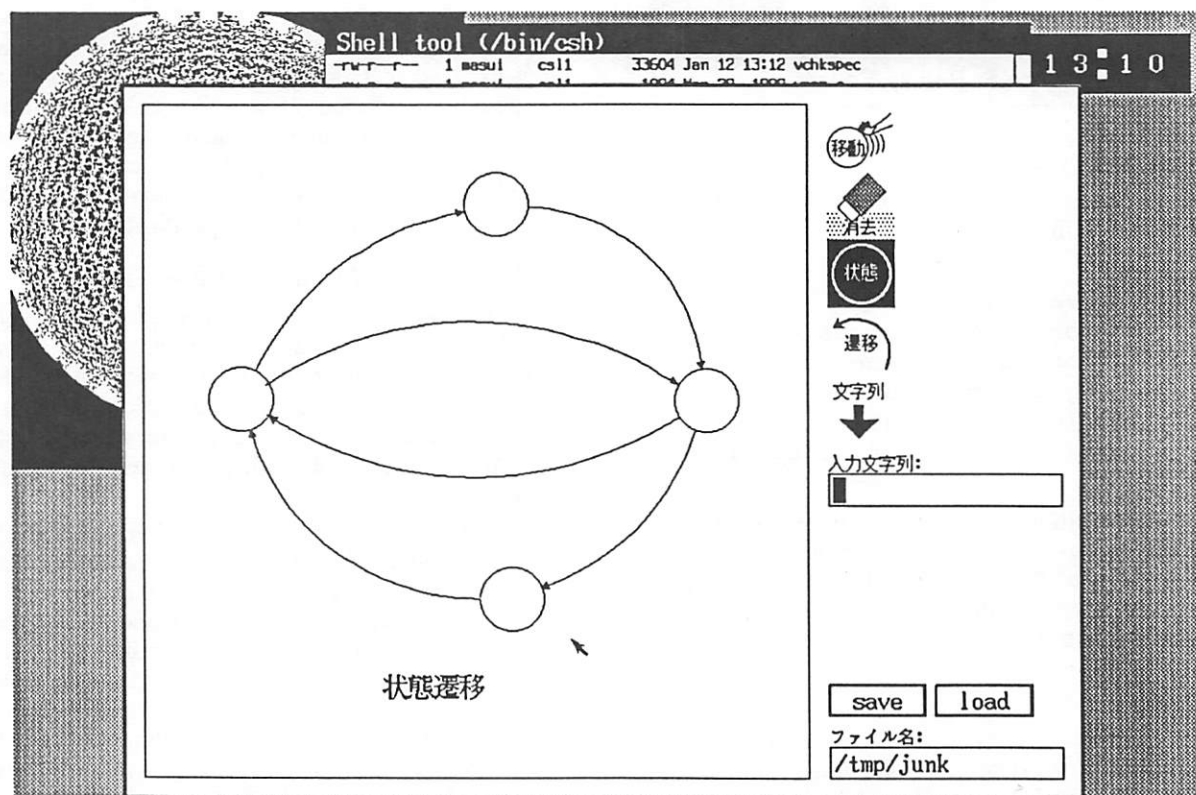


Figure 6: Drawing by Constraint

- Proceedings of SIGGRAPH* 20 pp. 241-248 (August 1986).
6. R. D. Hill, "Some Important Features and Issues in User Interface Management Systems," *Proceedings of SIGGRAPH* 21 pp. 116-120 (April 1987).
 7. D. Galernter, "Generative Communication in Linda," *ACM Transactions on Programming Languages and Systems*, pp. 80-112 (January 1985).
 8. N. Carriero and D. Galernter, "Linda in Context," *Communications of the ACM* 32 pp. 444-458 (1989).
 9. B. E. Casey and B. Dasarathy, "Modelling and Validating the Man-Machine Interface," *Software-Practice and Experience* 12 pp. 557-569 (1982).
 10. A. I. Wasserman, "Extending State Transition Diagrams for the Specification of Human-Computer Interaction," *IEEE Trans. on Software Engineering* 11 pp. 699-713 (August 1985).
 11. A. Borning, "ThingLab - A Constraint-Oriented Simulation Laboratory," no. SSL-79-3, Palo Alto, Calif. (1979).
 12. I. Sutherland, "SKETCHPAD A Man-Machine Graphical Communication System," *IFIPS Proceedings of the Spring Joint Computer Conference*, (January 1963).
 13. S. E. Hudson, "Graphical Specification of Flexible User Interface Displays," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 105-114 (November 1989).
 14. P. A. Szekely and B. A. Myers, "A User Interface Toolkit Based on Graphical Objects and Constraints," *Proceedings of the 1988 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 36-45 (August 1988).
 15. B. A. Myers, B. V. Zanden, and R. B. Dannenberg, "Creating Graphical Interactive Application Objects by Demonstration," *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pp. 95-104 (November 1989).
 16. J. Maloney, A. Borning, and B. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 381-388 (October 1989).

Toshiyuki Masui is currently a visiting researcher at the Center for Machine Translation in Carnegie Mellon University. He has been working on the development of a window system and user interface tools on UNIX workstations at SHARP Corporation (Osaka, Japan.) He received his B.S. (1984) and M.S. (1986) in Electric Engineering from University of Tokyo. He may be reached via electronic mail at either masui@cs.cmu.edu or masui@shpcsl.sharp.co.jp.



\$HOME MOVIE – Tools for Building Demos on a Sparcstation

Stephen A. Uhler – Bellcore

ABSTRACT

\$HOME MOVIE is a suite of tools for the capture, editing and playback of window system sessions on a Sun Sparcstation. It includes ISDN voice quality audio, video, and a VCR-like user interface. At any time while the window system is running, a recording may be started, generating a complete log or script that captures the changes to the display. Simultaneously, an audio script is generated, containing any verbal descriptions, or sounds present. Once these *recordings* have been made, they can be re-arranged, edited, annotated or set to music, using the \$HOME MOVIE sound and image editing tools. The resulting *movie*, can be played back on the display in real time, and thus provides a convenient way to document and demonstrate interactive software systems.

Introduction

Another demo. The boss walks in with another VIP just as you are starting to get some work done. Now you have to find a working version the program, crate the equipment into the other room, set it up, and pray everything holds together for the next ten minutes.

Presentation of high quality software demonstrations requires not only the appropriate hardware and software environment, but an expert user to manage the interface and often another person to explain what is happening. Existing demonstration methods include video-taped examples, which suffer from poor resolution and require special equipment, or special demonstration software with no audio capability.

\$HOME MOVIE is a system for the Sun Sparcstation that solves the problem of preparing demonstrations of interactive software systems. \$HOME MOVIE includes audio and video, with a television and VCR-like interface supporting pause, play, slow motion, fast forward, program selection and volume control. It requires no advance setup, and can be turned on at the spur of the moment. It is easy to add background music, or some video special effects and wind up with a snazzy self-contained demo.

How \$HOME MOVIE Works

Capturing the Demonstration.

There are two methods that can be used to capture a session. With the first method, *input saving*, the inputs to the application are saved, along with the times between inputs. To replay the session, the saved inputs are re-sent to the program that re-executes the session. With the same inputs as originally used, in the same order and relative timing, the visual results will be identical to the recorded session. In the second method, all changes to the display, a *display list* are saved, along with the

appropriate timing information. A stand alone driver program then interprets the display list to recreate the display images.

The *input saving* method has several advantages. For simple programs that require only keyboard and mouse input, the stored representation of the input can be made compact. In addition, capturing input can often be accomplished with no modifications to the program, by intercepting all input before it is sent to the application. Finally, since the demo'd program is actually running, arrangements can be made for the viewer to take over the execution of the demo, and actually run the program. This capability is quite useful for training and on-line documentation. JYACC [1] is an example of a commercial system that provides this type of capability. Another system Whimsy [2] uses the input saving technique in a windowing environment by capturing an applications inputs to the window system. Whimsy is intended more for testing than for demonstrations.

Unfortunately, the *input saving* technique has some limitations. For many systems it is difficult, or even impossible to capture the entire input to an application. In a network environment, there can be subtle interactions between other programs on the network, as well as non-repeatable interactions with the operating system or file system. To recreate the demo, not only would the demo program need to be re-run, but so would other programs on the network, and all referenced files, network hosts, and machine states; clearly a monumental task. Finally, the playback can't be sped up or slowed down, but can only run at the current speed of the program. Often it is in just this kind of transitory environment, with new software in development, that a demo is required. Recreating the entire state simply isn't possible.

With the *display list* method, which is used by \$HOME MOVIE, only the actual changes to the display are saved. None of the program input is required. Consequently, the program to be demo'ed is not needed for playback, and neither is the computing environment required to make the demo program run. An independent display list driver is used to recreate the applications display in real time. The demo is completely self-contained, can be distributed without compromising proprietary software, and can be run on a generic computing platform.

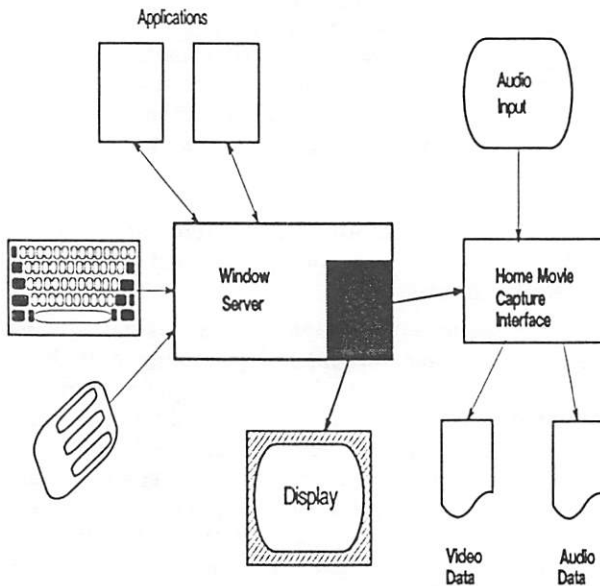


Figure 1: \$HOME MOVIE Recording Setup

How the Video Portion is Saved.

To be effective for capturing demos, the demo software needs to be un-obtrusive. The application must be completely unaware that its output is captured, and no changes to the application, or the way it is run can be required. In addition, the user should be able to turn the demo capture on or off at will, with no prior planning.

For \$HOME MOVIE to meet this goal, the window system server (not the application) is modified. All changes the window system makes to the display, along with timing information, are written onto a socket to be read by a separate process that saves the data in a file, the *video display list*. Figure 1 is a diagram of the demo capture setup.

The changes to the display are represented as the names of and arguments to the primitives that the window system uses to change the display. When a display primitive is invoked by the window server, a record of the invocation is generated. Recreating the display requires little more than reading the display list, then invoking the display primitives that were used by the window system to create the display in the first place.

The primary display primitive is a *bitblt* [3] (sometimes called a raster-op) which is used to change a rectangular set of pixels on the display. The *bitblt* operations are used to display images, print text, move the cursor, and update windows. The *bitblt* operations used by the window system often involve combining bitmaps (or pixmaps) that reside in memory with bitmaps in the frame buffer memory; the bitmaps that map to pixels on the display. For these operations it is necessary to keep track of not only the prior contents of the display, but to keep track of the contents of the memory bitmaps as well.

Although the entire demo session can be captured strictly with *bitblt* commands, some additional graphics primitives are used for improved efficiency. In addition to *bitblt*, these additional primitives are *points*, *lines*, and *circular arcs*. Still more graphics primitives, such as splines or polygons can be included in the display list as well, but are just as easily constructed out of the above primitives. The complete list of commands currently used by \$HOME MOVIE and generated by the window system is shown in Table 1.

The first three items, *Bitcopy*, *bitblt*, and *Point* are various flavors of *bitblt* commands. The *Data* item represents image data. *Arc* and *line* are additional drawing primitives added for efficiency. *Display* and *Free* are used for book keeping, *Time* is for time stamps and *Comment* data is ignored, and can be used by other programs that process the display lists.

Table 1: Display List Commands	
name	description
Bitcopy	<i>bitblt</i> - without source
Bitblt	<i>bitblt</i> - with source
Point	Draw a point
Data	Image data
Arc	Draw an elliptical arc
Line	Draw a line
Display	The display bitmap
Free	Free image data
Time	Time stamp
Comment	Comments

When the source or destination bitmap to a *bitblt* command is first referenced, its size and bit image are saved in the display list. When the playback program reads in the image for that bitmap, the image is cached for later use. The next time that bitmap is referenced, its image is already available in the video playback driver, so the image need not be repeated in the display list. For example, to display text in a window, the first time a character of a given font is referenced, the image of the entire font is saved in the display list. Every other character in the font is displayed by saving the *bitblt*

command required to copy that character from the already saved font image on to the display. The amount of memory required to cache the bitmaps varies with each application, but it is never more than was required of the server in the first place.

Code in the window system server keeps track of bitmap image changes, such as when a client application replaces one image with another, or when a bitmap is destroyed so that its contents are no longer required. In either case, a *bitmap free* command is saved in the display list, indicating a particular bitmap is no longer needed, permitting the display list driver to remove the image from its bitmap cache. The server then resends the new image data to the display list when required.

For most applications, the number of images that need to be saved in the display list is small, usually several fonts, icons, and cursors. All other display changes are made by combining these few images using *bitblt*, or other graphics primitives. When recording starts, the initial display image is saved in the display list, just as the first change to the display is about to occur. Each additional image is saved on the display list just as the first display primitive that references it is invoked. The window system server keeps a table of all images in use by

the server, so it can readily find those that are required for the demo.

In one sample \$HOME MOVIE session, a 13 minute demo of Superbook [4], a total of 29 images were saved in the display list. The image of the display when recording begins is saved, as well as image data for obscured windows that will be exposed during the demo. The rest of the images consist of cursors, fonts, icons, and graphics images specific to the application. The entire demo is created by performing *bitblt* transformations on the 29 images. Figure 2 shows the initial display of the Superbook demo, with the \$HOME MOVIE user interface above the top of the display. Figure 3 contains the images that were required to reproduce the rest of the demo. The first few images are the fonts used by Superbook, each saved in the display list as the first character in the font was referenced. The fonts are followed by various cursors and icons either by Superbook or the window manager. The next image is an illustration presented to the user by Superbook, whereas the last image contains the contents of a window that was obscured on the initial display. Table 2 lists a summary of the images and sizes required for this demo. The total stored size of the images was about 60 kilobytes.

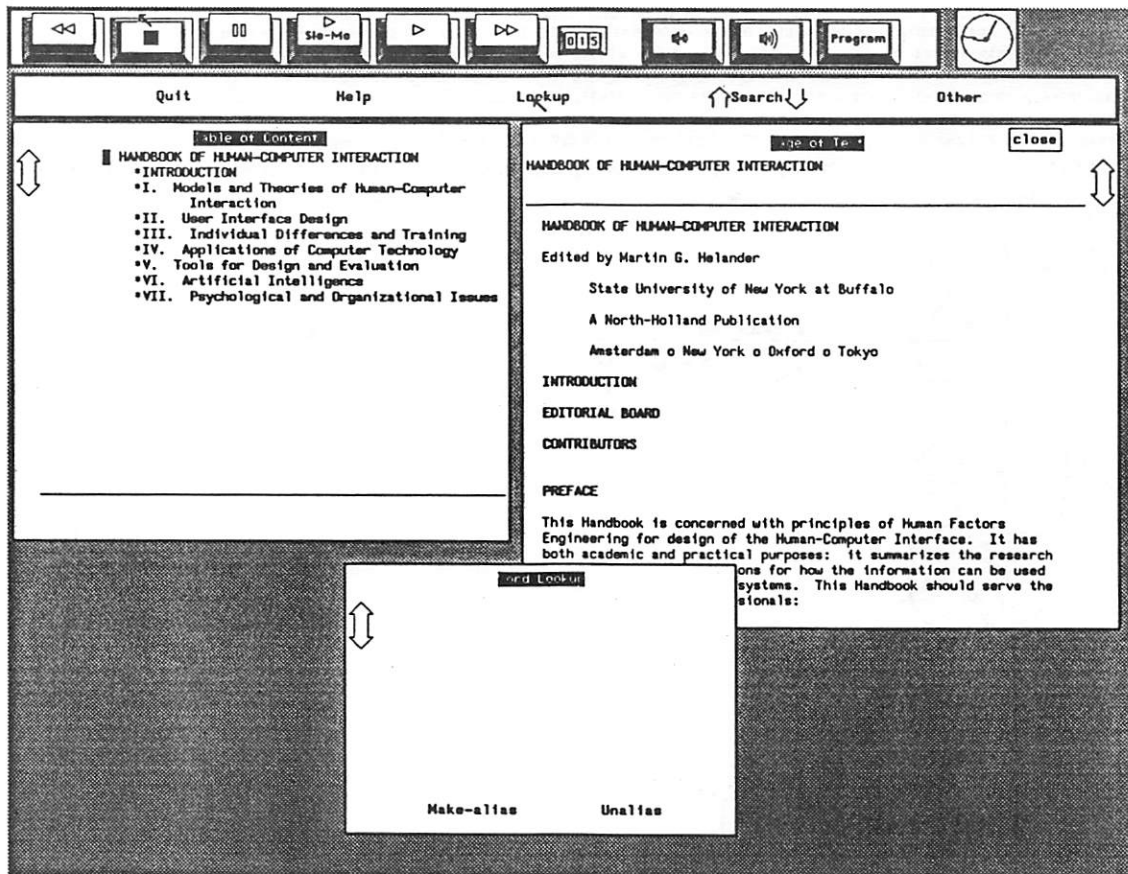


Figure 2: Initial Display of Superbook Demo

Because the display list stores low level *bitblt* and drawing primitives, the video data format is window system independent. It has no notion of fonts, cursors, images or windows; just operations that combine generic images. These are the same operations that are needed by all window systems that use a memory mapped model of the display. Consequently this method of recording demos is applicable to many window management systems, with no particular bias to any one in particular.

Table 2: image breakdown		
type	number	size (as displayed KB)
cursors	13	1
fonts	8	18
windows	3	83
images	2	51
initial display	1	130

The window system generates timing information periodically, typically in the main *dispatch* loop in the server. This timing information is saved as a time-stamp in the display list. A time-stamp is a 32 bit quantity that represents 100ths of seconds elapsed since the window system session began. This permits about eight months of display information to be

kept in a single display list. The display list driver program detects time-stamp overflow, thus permitting video scripts of almost unlimited duration. Absolute time information is used instead of time differences because it is easier to avoid rounding errors. It is also easy to alter the notion of time and play the display list back either faster or slower than it was originally generated. The video driver program understands a special *time offset* command that can be inserted into a display list at any point to add or subtract a fixed amount of time at any point in the display list without requiring the remaining time-stamps to be adjusted.

An arbitrary format was chosen for the display list data. In this format, each command consists of a 16 bit command identifier, followed by one or more arguments, as indicated by the command type. The display lists are normally stored in compressed form [5], and typically take less than 1000 bytes per second of demo.

In situations where the space consumed by the display list must be minimized, or where the video data needs to be transmitted in real time instead of saved in a file for later use, there are alternate data formats that vastly reduce the space required. A *bitblt* command is often similar to the previously

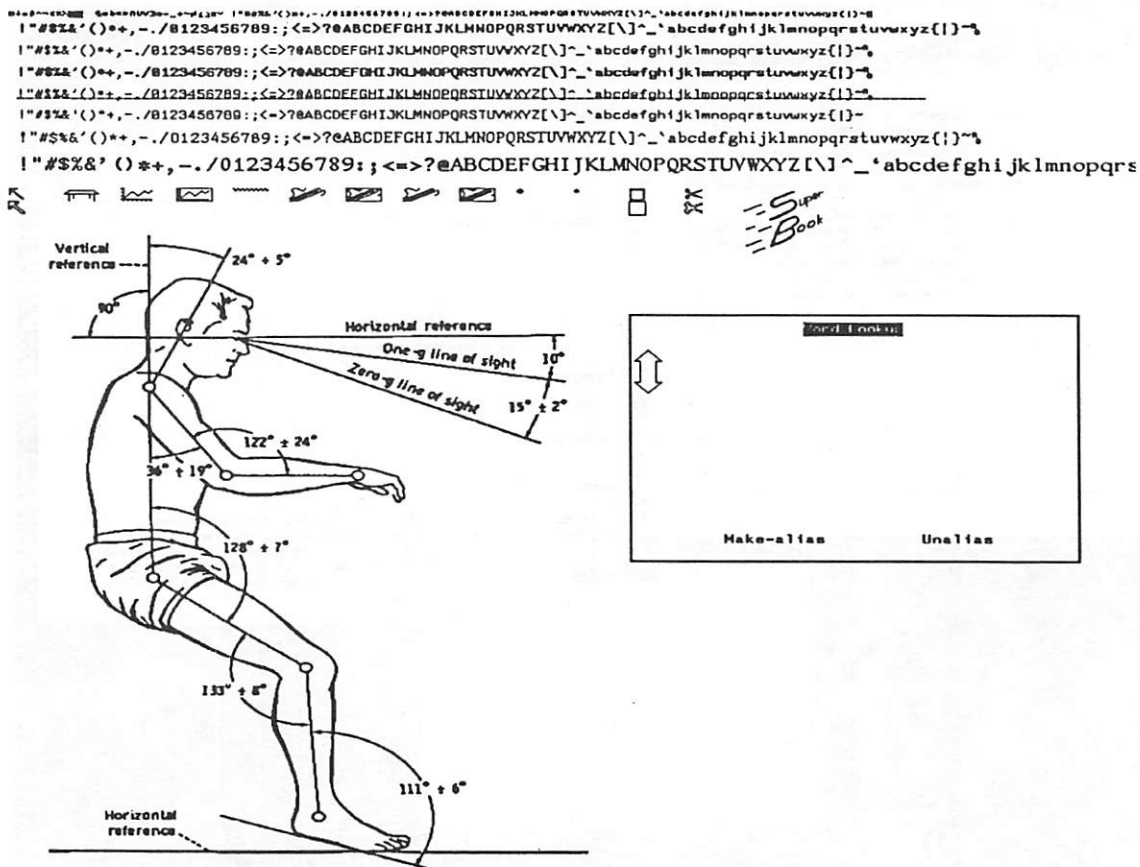


Figure 3: Collection of Saved Images used by The Superbook Demo

issued *bitblt* command. In 72% of the 89000 *bitblt* commands used in the Superbook demo, six or more of the nine arguments to the command were identical to the previous *bitblt* command. By choosing a display list format to encode differences from the previous command, substantial data compression can be achieved. In the extreme case, where ASCII terminal like output is prevalent, most *bitblt* commands can be encoded in a single byte. The usual case on the display in this instance is for the next character on the current line to be displayed. A source offset into the current font bitmap plus a destination offset on the display equal to the previous character width can be represented as a single character.

How the Audio Portion is Saved.

The audio portion of the demonstration is saved separately from the video script and the window system server. It is stored in a file in ISDN style μ -law format [6]. The μ -law format consists of 8000 8-bit samples per second. The audio can be voice, music, special effects, or other noise such as key-clicks or machine noise.

There are several reasons to keep the audio information separate from the video data. First of all, there is a large body of existing tools [7]

available to manipulate the audio data. These tools can be used as is.

The large data rate difference between the audio and video is a more compelling reason to keep the audio separate from the video portions of the demo. Whereas the audio portion of the script requires 8000 bytes per second, the average bandwidth for the video display list is only one tenth that. At well under a thousand characters per second, it is possible to transmit the video data in real time over common dial-up lines. Although this might not be beneficial for demos that require sound, it is invaluable for providing remote dial-up window system services, using the same tools as the required by \$HOME MOVIE.

How the Video and Audio Data are Synchronized.

Synchronization between the video and audio portions is maintained though the embedded timing information in the video script, and the fixed data rate format of the audio script. At regular intervals - about ten times per second, a timing mark is embedded in the video display list, representing the elapsed time in 100ths of seconds since the beginning of the script. That time, when multiplied by 80, represents the current byte offset in the audio file, thus

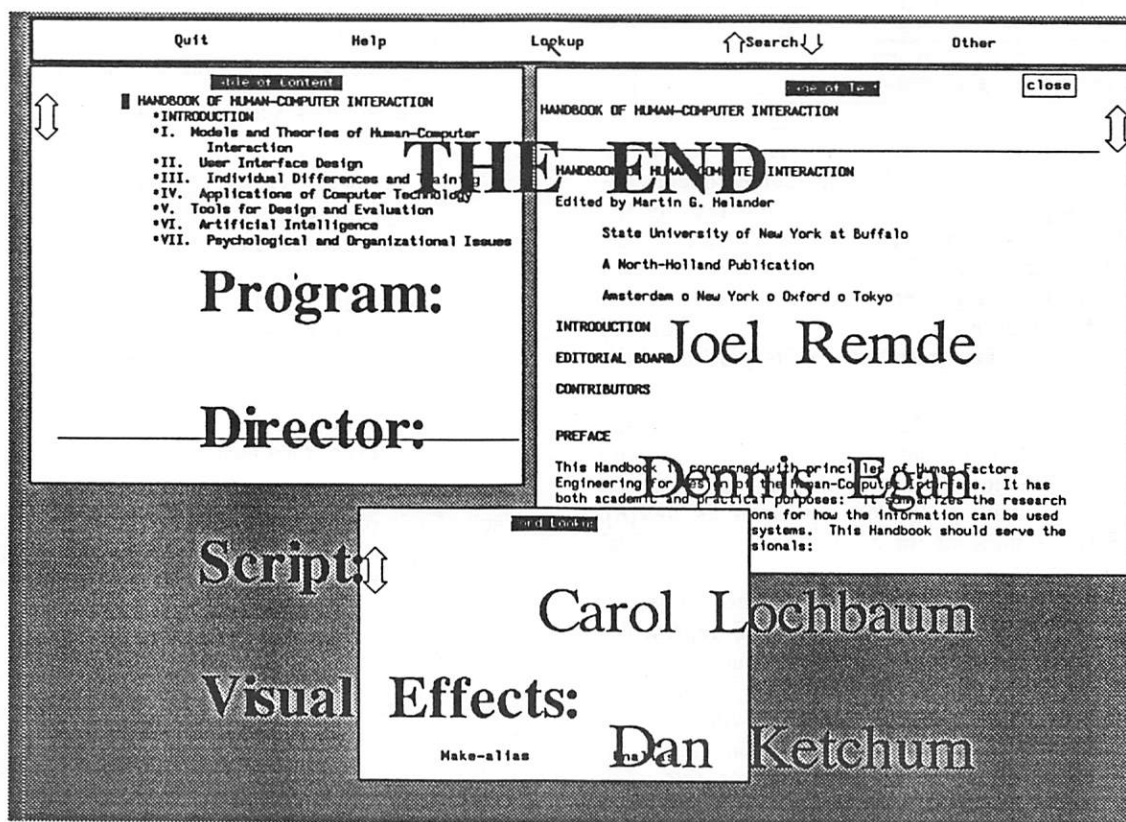


Figure 4: Sample Roll-A-Credit Output

maintaining synchronization to within 10ms, which is sufficient for demos.

Editing A Demonstration Script

Editing The Video portion

A Separate utility *Roll-A-Credit* was developed to generate text annotations, titles, and credits, which can be inserted into the video script files. *Roll-A-Credit* is a stand alone utility program that generates video display lists in the same format as the modified window system server. Words or phrases in one of several large fonts are animated by scrolling the text slowly on top of the current display. The initial and final position of each text phrase, the speed of scrolling, and the scrolling sequence is all under user control. The *Roll-A-Credit* display list is then inserted into a demo display list to effect the annotation. The input to *Roll-A-Credit* consists of one or more lines each containing four fields; 1) a font style and size, 2) text justification or position, 3) vertical offset from the previous line, and 4) the text to be animated. Often several *Roll-A-Credit* scripts are run consecutively to the same video display list, permitting different groups of text to be animated separately. Here is an excerpt from the *Roll-A-Credit* script used in the Superbook demo.

b-28	c	0	THE END
r-12	c	0	A Cog-Sci Video Production

The first line of the script causes *THE END* to be animated in a 28 point bold font, centered horizontally, and separated from the following text by the normal vertical spacing. The rate of animation, size of the drop shadows, and initial and final conditions are specified as arguments to the *Roll-A-Credit* command. Figure 4 shows a snapshot of *Roll-A-Credit*, taken from the credits portion of the Superbook demo.

The video script files can be converted to and from an ASCII representation using the program *to_ascii*. Once in ASCII, the display lists can be edited using standard UNIX tools such as *awk*[8]. Table 3 is a sample of the ASCII format. The initial character on the line is the command type; the remaining numbers are arguments to the command. Image data are saved in a hexadecimal representation. In the example above, the command characters *T*, *D*, *B*, and *L* stand for time-stamps, image data definition, *bitblts*, and lines respectively. Lines beginning with "." define the image data. The command character is followed by the arguments. For time stamps, it is the elapsed time in seconds. For *bitblts*, the most complex command, it is the destination bitmap number, the offset into the destination bitmap, the size of the rectangle, the *bitblt* function, the source bitmap number, and finally the source bitmap offset. The other command arguments are

defined similarly.

Table 3, sample ASCII data format

T	5.17								
D	13	32	16	1					
.	198C0000	198C0000	198C0000	39CC00					
.	0D9B8000	00180000	00180000	0000180					
.	00000180	00000000	00000000	00000000					
.	00000000	00000000	00000000	00000000					
.	00000000								
B	2	557	28	16	16	12	13	0	0
B	13	0	0	16	16	12	2	561	26
B	2	561	26	16	16	14	15	0	0
L	2	49	46	22	76	5			
L	2	49	46	19	73	5			
T	5.63								

To illustrate how one might edit a video display list, called *script.z* (the display list is stored compressed), suppose during the course of the demo, debugging output was accidentally turned on while displaying a line drawing in a window. The debugging text wrecked our drawing. To fix it on playback, we can delete the non-line drawing commands from the display list that were output on the drawing window. The following command would be run:

```
zcat script.z v to_ascii v
awk -f fix.awk v
to_binary v compress > new_script.z
```

The appropriate *awk* script, *fix.awk* is:

```
{
if ($1=="T" && $2<4 && $2>9)
    print # Not within the proper time range
else if ($1 != B && $1 != W)
    print # Not a bitblt command
else if ($2 != 2)
    print # Not destined for the display
else if ($3<46 vv $4<150)
    print # Not inside the window
else if ($3>850 vv $4>700)
    print # Not inside the window
}
```

Each clause of the *awk* script examines a line for the ASCII version of the command, and passes it through unaltered unless it is one of the commands targeted for deletion.

Each command in the display list consists of a line containing of the name of the command, followed by its arguments. The playback program, to be described later, has a mechanism to place marks in the display list under user control. The user can watch the demo, and add marks at any point. These marks can be later used to aid in editing the script. The program *to_binary* performs the inverse function, converting the script back to its binary form.

Images stored in the display list are represented in hexadecimal ASCII, similar in format to *od -x*. To facilitate easier editing, the images can be extracted from the display list and stored as separate files in a standard image file format. The images may then be viewed and edited using standard picture editing tools, and later re-combined with the rest of display list.

There is a library of canned video effect scripts that can be used to join demo scripts together. These canned scripts, when sandwiched between two disjoint display lists, provide for smooth transitions between the two. There are canned scripts for fading gradually from one display image to another, or fading to black or white, or pushing the current display image off the screen with the new one.

The canned scripts work by looking at the two scripts to be joined, calculating the final image displayed by the first script, and the initial image displayed by the second script, then constructing the primitives (*bitblt* commands) to generate the desired transition.

Audio Editing

There is a set of audio editing tools to cut, paste, and manipulate sections of audio. These tools include filters for AGC (automatic gain control), squelch, mixing, stretching and shrinking portions of the audio script. The *time_it* utility reads a video display list, and displays the elapsed time to the hundredth of a second at each mark and script merge. The corresponding audio editing tools are then used to extract the proper lengths of sound, to match up with the timings displayed.

The IMG (Incidental Music Generation) system [9] can be used to compose short pieces of music to use either as backgrounds under voice, or to call attention to annotations, titles, or credits. IMG knows how to compose a music in one of several different genres. The exact duration of the piece, as well as its tempo is specified by the user: IMG does the rest, producing a MIDI [10] file containing the composition. The MIDI file is then rendered in software, or fed to a MIDI synthesizer whose output is connected to the Sparcstation's audio input. Either method results in a μ -law rendition of the composition. The *shell* command

```
compose -121.4 grass v
play_midi > /dev/audio
```

composes and plays a complete 21.4 second blue-grass piece.

To add music to a Roll-A-Credit title or annotation sequence, *time_it* is used to determine the exact duration of the Roll-A-Credit animation. Then IMG is instructed to compose a piece of the proper length, that suits the mood of the demo. After instrumenting and synthesizing the piece, it is inserted into the audio track to accompany the

annotation.

Another use for IMG is to dub in background music underneath the narration in some parts of the demo. The demo is previewed, adding marks to the video script to indicate the beginning and ending of sections that need certain types of background accents. Either by using IMG, or clips of prerecorded music, the audio narration can be highlighted by mixing the music into the proper location to fit the audio narration or video display.

Playback

The playback portion of \$HOME MOVIE consists of three processes, the user interface, the video driver, and the audio driver. The user interface accepts mouse hits on buttons as commands from the user and translates them into ASCII commands that are sent to both the audio and video display processes. The audio and video display processes read and interpret the demo scripts, under the control of the commands sent by user interface process. A diagram of the playback setup is shown in Figure 5. A short shell script, *movie*, sets up the playback environment and starts the three playback processes.

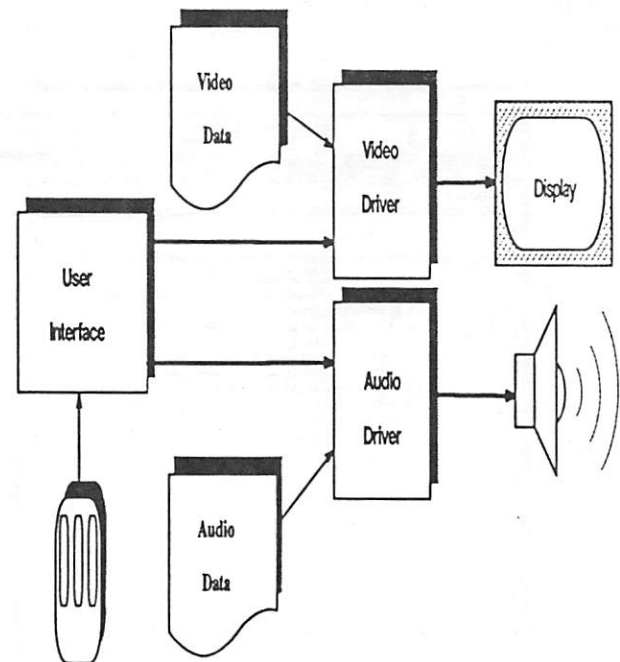


Figure 5: \$HOME MOVIE Playback Setup

User Interface

VCR, the primary user interface to \$HOME MOVIE, simulates the functions found on a video cassette recorder. Figure 2 shows a picture of the user interface, at the top of a demo in progress. VCR provides a mouse activated button interface to the \$HOME MOVIE playback system. From left to right it has buttons for rewind, stop, pause, slow motion, and fast forward. Following fast forward is

a tape counter, volume down and volume up. Finally at the right edge is a program button. Except for the program button, the interface works just like a standard VCR. The playback start up script normally starts in just the top inch of the display, with the VCR program running. The remainder of the display is used by the video driver for the demo.

Using the program button on VCR users can step through a sequence of *tapes*, choosing the one they wish to play. VCR reads a startup script that maps each tape name shown on the *program* button into a list of one or more demo scripts which are played consecutively when that name is selected.

Since the design of the playback system is modular, the various parts can be easily interchanged. By replacing the user interface by a command file, repeated playback of a sequence of demo scripts results in completely unattended demos.

Video Playback

The video driver accepts commands from the user interface and plays back the video script. Normally the video driver writes directly to the display, except for the top inch occupied by the user interface.

During normal *play* operation, the video driver reads and processes display commands from the display list generated when the demo was first created. The time-stamps embedded in the display list are compared against the real time elapsed since the beginning of the script. Whenever the script elapsed time is greater, the video driver sleeps for the difference, thus recreating the pace of the original demo. If the user selects *fast forward* or *slow motion* on the user interface, the script elapsed time is multiplied by a constant other than unity. The effect is to speed up or slow down the notion of time, permitting playback either faster or slower than the original demo.

Audio Playback

The audio driver, like the video driver, also accepts commands from the user interface. In the current implementation, those commands are passed from the user interface through the video driver, so the playback mechanism can easily be started as a pipeline by the shell. the commands tell it to read the audio data from the appropriate point in the audio file, and send it to `/dev/audio` to be played out the speaker. As the video playback is stopped, started, speeded up or slowed down, the audio driver

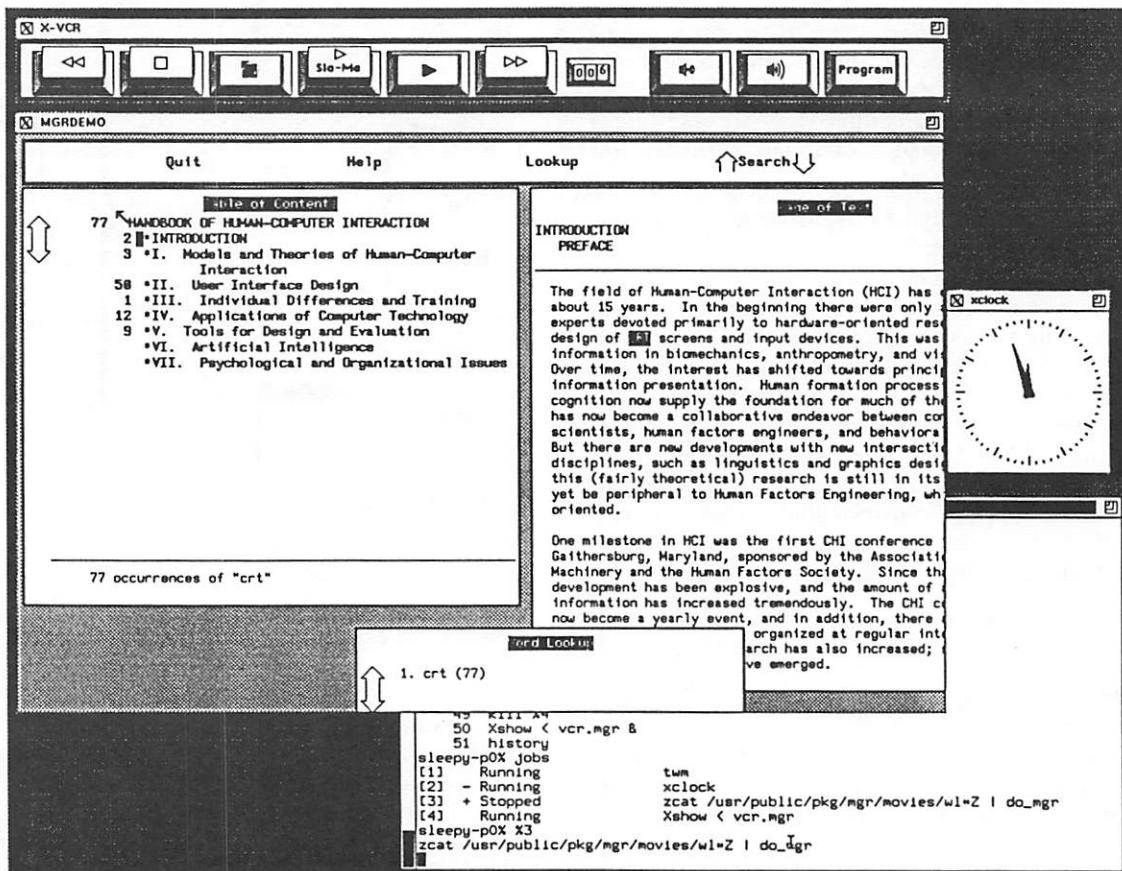


Figure 6: Superbook Demo in an X Window

receives synchronization commands from the user interface so it can determine which byte of the audio file should currently be coming out of the speaker.

When slow motion or fast forward is selected, the audio track is processed to run slower or faster without changing the pitch of the sound track, and can maintain intelligibility over a wide range of playback speeds. This is accomplished by either eliminating or duplicating groups of samples, then smoothing the edges where the groups abut. The fraction of samples removed or duplicated determines how fast the audio track is speeded up or slowed down.

Design Tradeoffs

The low level format for saving the display data was chosen to be window system independent, and requires only a simple driver program to play back the script.

The video and audio tracks are kept separate, in spite of potential synchronization problems and editing difficulties, so the tools that manipulate the data are simpler, and the video portion of the system runs unchanged for systems with no audio capability.

Implementation Considerations

The current version of \$HOME MOVIE was produced by modifying the MGR [11] window system to send the video display list information out a socket, about 300 lines of C code. The X11 [12] version of \$HOME MOVIE is under development, and uses the same strategy.

There are several ways to play back the demo. Normally, playback is made to the raw display. The video driver program is completely self-contained, and consists of 1600 lines of C code. About 1000 lines comprises the *bitblt* engine, the remainder reads and interprets the display list and user interface commands.

The video scripts can be played back into an X window. The X version of the video driver program, using Xlib calls, is a 1000 lines of C code. For X video driver, the display lists are played back in a window instead of the entire display. This allows the \$HOME MOVIE system to be used in the context of another application, such as providing on-line animated help. Figure 6 shows a portion of the Superbook demo playing back in an X window.

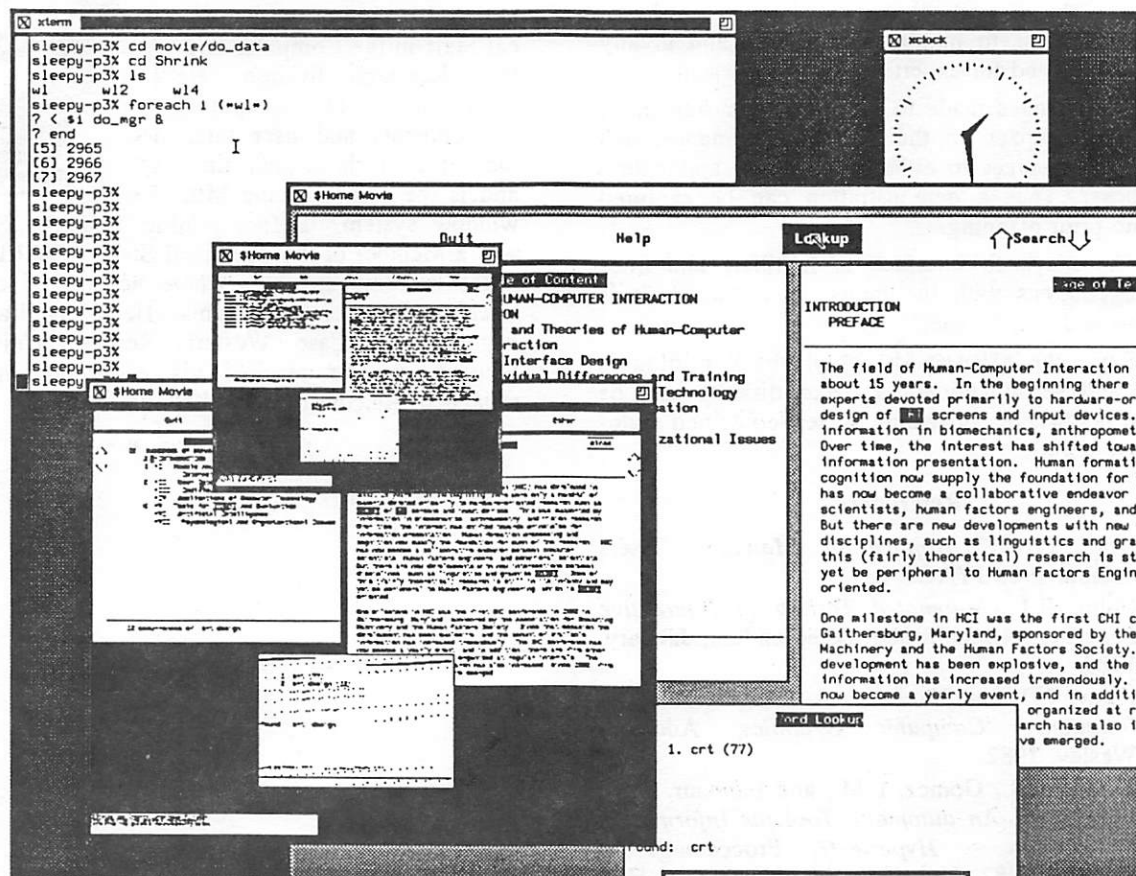


Figure 7: Shrunken Playback in X

The playback can be shrunk to a smaller size than the original recording, permitting playback in less than display-sized windows. This is a direct consequence of the geometrical nature of the video display list format. The drawing commands are scaled simply by scaling their coordinates. Only the images need any significant processing, and are easily reduced in size by integral multiples, although with a corresponding loss of resolution. Figure 7 shows a portion of the Superbook demo playing back in three separate windows; full size, half size, and quarter size.

The audio and video playback portions of \$HOME MOVIE are played by separate processes, so even though synchronization data is kept to within a hundredth of a second, due to the granularity of the UNIX scheduler, the audio and video synchronization has considerably greater variance. Packaging the video and audio playback in the same process would ameliorate this problem to some extent, but at the cost of some flexibility.

Conclusions

As a test case, a 13 minute demonstration of the SuperBook hypertext system was prepared. The video display list averages 791 bytes per second, whereas the audio requires a constant 8000 bytes per second. The Superbook movie has been shown dozens of times to hundreds of people and greatly reduces the need for expert users to be present.

The changes made to the window system have a minimal impact on the server performance, and require no changes to either the user or application interfaces. Thus a demonstration can be captured with no prior planning.

The playback interface is familiar, and once pushing buttons with the mouse is mastered, it is obvious and easy to use.

Since the \$HOME MOVIE playback portion is small and self-contained, a *demo* diskette can be mailed anywhere, providing a self-contained autonomous demo.

References

- [1] JAM JYACC *Application Manager*, Users manual 1988 JYACC Inc.
- [2] Cohn, R.J., *Automated Testing of Interactive Programs*, Unpublished memorandum, January, 1987.
- [3] Foley, J.D. and VanDam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1982.
- [4] Remde, J.R., Gomez, L.M., and Landaur, T.K., *SuperBook: An automatic Tool for Information Exploration -- Hypertext?*, Proceedings of Hypertext '87. University of NC, Chapel Hill, 1987 pp. 307-323
- [5] Welch, T.A., *A Technique for High*

Performance Data Compression, IEEE Computer, vol. 17, no. 6 (June 1984), pp. 8-19.

- [6] Members Technical Staff, Bell Telephone Laboratories, *Transmission Systems for Communications*, 1959.
- [7] Langston, P.S., *UNIX Midi Manual, Sections I, III, & V*, Internal Bellcore Technical Memorandum TM ARH-015440, November, 1989
- [8] Aho A.V., Kernighan W.K., Weinberger P.J., *The AWK Programming Language*, Bell Telephone Laboratories, 1988
- [9] Langston, P.S., *PellScore, An Incident Music Generator*, Internal Bellcore Technical Memorandum TM-ARH-016281, February, 1990
- [10] DeFuria, J.S. and Scacciaferro, *MIDI Programmers Handbook*, M&T Publishing Co., 1989
- [11] Uhler, S.A., *MGR - C Language Application Interface*, Bellcore Internal Technical Memorandum TM-ARH-010796, December, 1988
- [12] J. Gettys, R. Newman, T. Della Fera, *Xlib - C Language X Interface*, January 1986.

Stephen Uhler joined Bell Communications Research at its inception in 1984, where he is a Member of the Technical Staff in the Computer Systems Research division. He has worked on computing environments and user interfaces for much of that time, and is the author of the MGR window system. Before joining Bellcore, Stephen was a Member of the Technical Staff at Bell Laboratories in Whippany N.J. where he worked on user interface management systems. He received an M.S. degree from Case Western Reserve University. Stephen can be reached via electronic mail at: sau@bellcore.com or uunet!bellcore!sau.



Awk As A Major Systems Programming Language

Henry Spencer – University of Toronto

ABSTRACT

Even experienced Unix programmers often don't know *awk*, or know it but view it as a counterpart of *sed*: useful "glue" for sticking things together in shell programming, but quite unsuited for major programming tasks. This is a major underestimate of a very powerful tool, and has hampered the development of support software that would make *awk* much more useful. There is no fundamental reason why *awk* programs have to be small "glue" programs: even the "old" *awk* is a powerful programming language in its own right. Effective use of its data structures and its stream-oriented structure takes some adjustment for C programmers, but the results can be quite striking. On the other hand, getting there can be a bit painful, and improvements in both the language and its support tools would help.

Introduction

There is a very large gap between the UNIX shell and the usual UNIX programming language—C—in power and ease of use. Shell programming is easy and the variety of available high-level primitives (programs) is large, but if your needs do not match the available primitives, you are basically out of luck: the low-level primitives are skimpy and extensive use of them is very inefficient. C, by contrast, provides a fairly full set of very efficient low-level¹ primitives, which are tricky and dangerous to use and often require extensive programming for operations that are trivial in the shell. Programmers need a simple programming language that can fill this gap: one that is easy and safe to use for simple jobs, while being versatile enough to cope with the unexpected, and acceptably efficient for undemanding tasks.

Awk [2] is a good choice for this purpose, and indeed it is widely used for small programs and for building otherwise-unavailable primitives for shell programs. It is available on nearly every UNIX system (and a good many UNIXnon- systems too), and with the exception of occasional niggling details, *awk* programs are highly portable.

Unfortunately, this widespread use as "glue" has hampered acceptance of *awk* as a serious programming language. Worse, a vicious circle has developed: the lack of appreciation of *awk*'s uses for serious programming has prevented development of the support tools that would make it more obviously viable. The combination has given *awk* a reputation as being unsuited to major programming

tasks. Recent experiences have convinced us, with reservations, that this reputation is undeserved.

A Note On Terminology

There are two major variants of *awk* currently in circulation: the original [2], released with UNIX Version Seven in 1979, and "new *awk*", featured in the *awk* book [3] and the model for most modern implementations.

Unfortunately, although new *awk* is a considerably superior language, its availability is somewhat limited as yet: many UNIX vendors still ship the old *awk*, the independently-available implementations either cost substantial amounts of money or come with troublesome licences, and there are vast numbers of old systems (with old *awks*) which will be in active use for years to come. As a practical consideration, *awk* programs intended to be portable must be written in old *awk*.

This paper will follow common practice and refer to new *awk* as *nawk*; unqualified references to "awk" refer to old *awk*.

A Learning Experience

One reason why *awk*'s acceptance has been slow is that, for a C programmer, it takes getting used to. Programmers who rarely use it, or use it only as "glue", seldom learn it well enough to get much appreciation of its capabilities. As with the UNIX shell and its vast array of "filter" programs, becoming a really proficient *awk* programmer takes time and experience, because not all the uses of its more non-C-like features are obvious at first glance.

In this regard, some experiences with a couple of major *awk* projects are of interest.

¹C's libraries are a disgrace [1], and improvements there would help a great deal in making it a more livable programming language, but there is little sign of this happening.

Text Formatting With Awk

Long long ago, in a UNIX community that certainly seems far away by modern standards, all UNIXes came with *nroff* (and perhaps *troff* as well, if you were lucky). This made *nroff* source the de-facto exchange format for complex text, notably manual pages and other documentation for software. There were some difficulties, notably the variations in macro packages, but aside from that, one could confidently send a friend program documentation and expect him to be able to read it.

Unfortunately, this happy state of affairs broke down when the UNIX text formatters were "unbundled". The theory is that only those who need the formatters will buy them. The practice is that many who need them, to varying degrees, do not have them and cannot afford them. Support for complex output devices and elaborate formatting is arguably not that common a need, but almost everybody needs a way of printing manual pages.

The "unbundled" answer to this is to provide preformatted pages, but these are not much better than printed copies. It is generally impossible to modify them for purposes like noting bugs and local changes, creating indexes into them is difficult at best, and adding your own pages to document local software is impossible without the text formatter.

The C News project [4] ran into this nuisance in connection with distributing documentation. It wasn't hard to invent a little macro package covering the forms used in our simple documentation, and this bypassed the problems of differing macro packages, but the complete lack of formatters at some sites was harder. We didn't want to distribute preformatted pages: they are a nuisance to generate, often problematic to transmit because of very long lines and control characters, and impossible to patch without spectacularly bloating *diff* listings.

In the course of moaning about this problem and cursing those responsible for unbundling, the idea of building a simple text formatter² came up. It looked like quite a bit of work in C, so it got shelved. Then the idea arose: could it be done in *awk*? The more this was investigated, the more promising it looked, for our limited purposes at least. The result was *awf*³.

²Obviously, this does not solve the whole problem, since people on unbundled systems are still stuck with all the preformatted pages from their supplier. However, Geoff Collyer's experimental manual-page "decompiler", *nam* (to appear on Usenet before this paper is published, barring disasters), deals with that issue.

³Actually its original working name was *off*, meant to suggest a rather drastic subset of *nroff*/*troff*, and also the somewhat repellent concept. The current name seemed superior, however, and the expansion "Amazingly Workable Formatter" was invented to justify it.

Awf [5] is a simple text formatter, emulating *nroff* -man or a subset of *nroff* -ms⁴, written entirely in (old) *awk*. It's seriously slow but has proved portable well beyond the author's expectations, with a VMS port published (in *comp.os.vms*) and a wide variety of other uses reported. It can handle almost any -man manual page and simple -ms documents. The output formatting, for a dumb terminal/printer, is nearly indistinguishable from *nroff*'s.

Awf has three passes: macro expansion (including parameters, macro nesting, and limited conditionals), command interpretation (including fonts, non-ASCII characters, and limited hyphenation), and page setting (somewhat underdeveloped by comparison, but does centering, margin adjustment, and river avoidance in particular). The sizes, in lines and bytes, are:

Pass	Lines	Bytes
1	212	5179
2	588	12311
3	332	9502

In addition, each macro package has a small piece of package-specific *awk* code, typically 30-40 lines, that is incorporated into the second pass before it is run. This typically handles a few details that cannot easily be expressed as macros in *awf*'s rather limited subset of the *nroff* input language.

Awf actually started out with much more of the semantics of the macro packages imbedded in *awk* code, but as it grew and the number of implemented *nroff* primitives rose, most of the complexity moved out into actual macro packages. The biggest problem in *awf* development was deciding where to call a halt, since simple *nroff* features often required only a few lines of code each. The code was remarkably easy to work with, even compared to normal C code⁵.

As mentioned above, *awf* is pretty slow, but it's fast enough to be practical when there are no alternatives! Formatting its own manual page, about 2.5 pages of moderate complexity (by manual-page standards), takes about 90 seconds of CPU time on a Sun 3/180. The second pass accounts for about half of this, with the remainder split fairly evenly between the first and the third. The bulk of it is user CPU time, although system time for the I/O is not negligible. A determined effort could probably speed this up somewhat, although almost certainly not enough to compete with *nroff* (which takes about 7 seconds to do the same text).

⁴Note that, despite occasional mistaken reports, *awf* is not a general-purpose *nroff* emulator. It uses its own simple versions of these two macro packages, and implements very little of the full generality of *nroff*.

⁵The insides of *nroff* cannot be described as normal.

Lessons From Awf

Awk strongly encourages programming with a "stream" orientation, in which the input is a stream of text lines, processed one at a time to produce output lines. Although *nawk* has added some features to permit multiple inputs, the overall structure of *awk* programs remains dominated by *awk*'s pattern+action paradigm. One can analyze lines in much the way a C program would, with one large action using *ifs* to analyze the input, but in practice *awk* code is both simpler and more readable if it is broken into relatively small actions, using *awk*'s powerful and flexible patterns to decide when each is applicable. In the case of *awf*, this structure is weakly visible in the first pass, very strongly visible in the second pass, and essentially nonexistent—inappropriately so—in the third pass.

The first pass handles macro definitions with patterns and small actions, but macro expansion and conditionals are a different story. They are processed by a single huge loop with a complex mass of conditionals (thankfully, not nested much) inside it. *Nawk*'s user-defined functions could make the control structure more readable, by separating iteration through a macro body from recursion to deal with nesting. However, the control structure could also be "flattened out" using the existing pattern+action structure of *awk*, were it not for a second, rather more subtle, problem: although a simple assignment to *\$0* will change the input line that *awk* matches patterns against, there is no way to tell *awk* to start matching all its patterns over again against the existing *\$0*. The main loop of the first pass is basically an *awk* program in miniature, with each iteration examining the "input" line for various conditions requiring attention, manipulating a simple stack for nesting, and finally preparing the next "input" line.

The dominant structure of the second pass is pattern+action, interrupted by a couple of large actions with imbedded loops, one to scan a line of text for things requiring special attention, the other to evaluate the terms of an arithmetic expression. Expression evaluation would probably be better as recursive procedures in *nawk*, but input scanning would fit a generalized version of the *awk* paradigm very nicely. This would have to be defined as a control structure in the language, perhaps analogous to C's *switch*.

The third pass *could*, in retrospect, be rewritten to make much heavier use of pattern+action. At present it is mostly one huge monolithic action for historical reasons: its structure changed repeatedly during its evolution, and the possibilities for cleanup present in the final version were not obvious in the earlier ones.

As a minor issue of pattern+action structuring, very often most actions completely deal with the input line and hence end with *next* to make *awk* pick up new input. The analogy to C's *switch* statement appears again, because subtle bugs can appear if one of those *nexts* is left out and later patterns rely on not seeing input that matched earlier ones. The situation is more difficult than in C, however, because the greater generality of *awk* patterns make such "fallthrough" much more useful.

Apart from the overall structure, the one other conspicuous lesson from the innards of *awf*'s passes is that old *awk*'s regular-expression primitives are seriously inadequate. It was repeatedly necessary to resort to *substr* loops because, while it is easy to determine whether a regular expression matches a string, it is not possible to determine *how much* of the string it matches. The *substr* loops are not only much messier to write and read, they are also a lot slower than doing the whole scan or substitution in a single primitive. *Nawk* has addressed this problem fairly well, and one can only hope that it becomes more widely available.

A more subtle issue of how *awk* (and in particular, old *awk*) affects programming is the multi-pass structure itself. The structure has its advantages: it does a very good job of "separation of concerns", making the individual passes much more comprehensible than their interwoven counterparts in *nroff*. On the other hand, it requires a lot of I/O activity to emit, pass, and parse the intermediate languages. More subtly, it makes feedback loops between the passes impossible. For example, a conditional statement (in the first pass) cannot do a comparison on a string variable, because it's the second pass that does the character-by-character dissection of lines needed to implement string-variable substitution. Even when feedback loops do not interfere, defining good pass boundaries can be difficult.

Awk's stream orientation and pattern+action structure is very convenient when the problem can be broken down into fairly independent passes, but gets in the way otherwise. Unfortunately, in old *awk* there is no real alternative if one wants to avoid mass duplication of code. The pass structure of *awf*, for example, was heavily influenced by the requirement that all processing of each type of construct be in one place to avoid duplication. If a particular type of data can appear from one of several sources (e.g. text lines from original input or from inside macros), life is often simpler if those sources are merged into one by putting a pass break after them, so that the destination sees a single stream of input. A minor example of this is that the second pass of *awf* emits the equivalent of ".ne 999" at the end of input, so that the third pass need not duplicate its complex end-of-page code in its end-of-input action.

Generating Parsers In Awk

Awk was successful enough to inspire a somewhat more whimsical experiment: a parser generator written in *awk*. The input language, dubbed AASL (Amazing Awk Syntax Language), was somewhat inspired by S/SL [6]; it is a simple notation for top-down parsing, analogous to syntax charts (aka "Railroad Normal Form") or to the code skeleton of a recursive-descent parser. As an example, the AASL specification for a simple form of arithmetic expressions could be written like this:

```
expr: term { "+" term ? } ;
term: factor { "*" factor ? } ;
factor: ( number | "(" expr ")" ) ;
```

There are also provisions for lookahead, control of error recovery, and insertion of semantic actions.

The implementation of AASL was fairly straightforward, with AASL itself used to describe its own syntax. An AASL specification is compiled into a table, which is then processed by a table-walking interpreter. The interpreter expects input to be as tokens, one per line, much like the output of a traditional scanner. A complete program using AASL (for example, the AASL table generator) is normally three passes: the scanner, the parser (tables plus interpreter), and a semantics pass. The first set of tables was generated by hand for bootstrapping.

Apart from the minor nuisance of repeated iterations of language design, the biggest problem of implementing AASL was the question of semantic actions. Inserting *awk* semantic routines into the table interpreter, in the style of *yacc*, would not be impossible, but it seemed clumsy and inelegant. *Awk*'s lack of any provision for "compile time" initialization of tables strongly suggested reading them in at run time, rather than taking up space with a huge BEGIN action whose only purpose was to initialize the tables. This made insertions into the interpreter's code awkward.

The problem was solved by a crucial observation: traditional compilers (etc.) merge a two-step process, first validating a token stream and inserting semantic action "cookies" into it, then interpreting the stream and the cookies to interface to semantics. For example, *yacc*'s grammar notation can be viewed as inserting fragments of C code into a parsed output, and then interpreting that output. This approach yields an extremely natural pass structure for an AASL parser, with the parser's output stream being (in the absence of syntax errors) a copy of its input stream with annotations. The following semantic pass then processes this, momentarily remembering normal tokens and interpreting annotations as operations on the remembered values. (The semantic pass is, in fact, a classic pattern+action *awk* program, with a pattern and an action for each annotation, and a general "save the value in a variable" action for normal tokens.)

The one difficulty that arises with this method is when the language definition involves feedback loops between semantics and parsing, an obvious example being C's `typedef`. Dealing with this really does require some imbedding of semantics into the interpreter, although with care it need not be much: the in-parser code for recognizing C `typedefs`, including the complications introduced by block structure and nested redeclarations of type names, is about 40 lines of *awk*. The in-parser actions are invoked by a special variant of the AASL "emit semantic annotation" syntax.

A side benefit of top-down parsing is that the context of errors is known, and it is relatively easy to implement automatic error recovery. When the interpreter is faced with an input token that does not appear in the list of possibilities in the parser table, it gives the parser one of the possibilities anyway, and then uses simple heuristics to try to adjust the input to resynchronize. The result is that the parser, and subsequent passes, always see a syntactically-correct program. (This approach is borrowed from S/SL and its predecessors.) Although the detailed error-recovery algorithm is still experimental, and the current one is not entirely satisfactory when a complex AASL specification does certain things, in general it deals with minor syntax errors simply and cleanly without any need for complicating the specification with details of error recovery. Knowing the context of errors also makes it much easier to generate intelligible error messages automatically.

The AASL implementation is not large. The scanner is 78 lines of *awk*, the parser is 61 lines of AASL (using a fairly low-density paragraphing style and a good many comments), and the semantics pass is 290 lines of *awk*. The table interpreter is 340 lines, about half of which (and most of the complexity) can be attributed to the automatic error recovery.

As an experiment with a more ambitious AASL specification, one for ANSI C was written. This occupies 374 lines excluding comments and blank lines, and—with the exception of the messy details of C declarators—is mostly a fairly straightforward transcription of the syntax given in the ANSI standard. Generating tables for this takes about three minutes of CPU time on a Sun 3/180; the tables are about 10K bytes.

The performance of the resulting ANSI C parser is not impressive: in very round numbers, averaged over a large program, it parses about one line of C per CPU second. (The scanner, 164 lines of *awk*, accounts for a negligible fraction of this.) Some attention to optimization of both the tables and the interpreter might speed this up somewhat, but remarkable improvements are unlikely. As things stand—in the absence of better *awk* implementations or a rewrite of the table interpreter in C—it's a cute toy, possibly of some pedagogical value, but not a useful production tool. On the other hand, there

does not appear to be any *fundamental* reason for the performance shortfall: it's purely the result of the slow execution of *awk* programs.

Lessons From AASL

Many of the earlier comments on results from *awf* also apply to AASL. The scanner would be *much* faster with better regular-expression matching, because it can use regular expressions to determine whether a string is a plausible token but must use *substr* to extract the string first. *Nawk* functions would be very handy for modularizing code, especially the complicated and seldom-invoked error-recovery procedure. A *switch* statement modelled on the pattern+action scheme would be useful in several places.

Another troublesome issue is that arrays are second-class citizens in *awk* (and continue to be so in *nawk*): there is no array assignment. This lack leads to endless repetitions of code like:

```
for (i in array)
    arraystack[i ":" sp] = array[i]
```

whenever block structuring or a stack is desired. *Nawk*'s multi-dimensional arrays supply some syntactic sugar for this but don't really fix the problem. Not only is this code clumsy, it is woefully inefficient compared to something like

```
arraystack[sp] = array
```

even if the implementation is very clever. This significantly reduces the usefulness of arrays as symbol tables and the like, a role for which they are otherwise very well suited.

It would also be of some use if there were some way to initialize arrays as constant tables, or alternatively a guarantee that the *BEGIN* action would be implemented cleverly and would not occupy space after it had finished executing.

A minor nuisance that surfaces constantly (in *awf* as well as AASL) is that getting an error message out to the standard-error descriptor is painfully clumsy: one gets to choose between putting error messages out to a temporary file and having a shell "wrapper" process them later, or piping them into "cat >&2" (!).

As with *awf*, the multi-pass input-driven structure that *awk* naturally lends itself to produces very clean and readable code with different phases neatly separated, but creates substantial difficulties when feedback loops appear. (In the case of AASL, this perhaps says more about language design than about *awk*.)

Support Tools

Although there are a few places where *awk* could really use language improvements, by far its biggest shortcomings right now are problems of implementation and support. The language itself is,

as demonstrated by some of the above, not that big a barrier to writing major programs. Unfortunately, aspiring *awk* programmers get very little help from their environment. The usual *awk* implementation is an interpreter, well-suited to small "glue" programs and to fast-turnaround testing but ill-adapted to production use with large programs. Its performance for substantial computing is poor, and its facilities for debugging and tuning are nonexistent (to the point of *awk* being notorious for not even being able to produce intelligible complaints about syntax errors, although *nawk* is much better).

Surprisingly, the first support tool that *awk* would benefit from is a precise language specification. Portability of *awk* programs is annoyingly hampered by small differences in fine points of syntax, points which are not resolved by the rather informal specifications published to date. For example, putting a slash into a character class in a regular expression simply cannot be done in a portable way, because the obvious

```
/...[.../].../
```

is a syntax error in some implementations, and the fix

```
/...[...\/].../
```

puts backslash into the character class in others. For another example, while all implementations agree that a regular expression by itself is a valid pattern, implicitly matching against \$0, there is substantial disagreement on whether this form of pattern can be combined with others by using && and ||; some *awks* will take the combined form only if the match against \$0 is made explicit. Yet another: the original *awk* implementation was happy to accept multiple pattern+action pairs on one line, which was very convenient for trivial "glue" programs, e.g.

```
awk '{ x += $1 } END { print x }' $*
```

but some of the more recent implementations have retroactively declared this illegal, based on vague implications in the *awk* manual (still vague in the *nawk* book) that each action should be followed by a newline. And so on. A precise, nit-picking specification of the *exact* syntax of the language would aid *awk* portability by eliminating this senseless diversity.

The next, and much more obvious, *awk* tool of importance would be a fast implementation. For example, AASL would be perfectly viable, at least for small-scale use, if its interpreter were not so slow. It's hardly surprising that an interpreter implemented in an interpreter is a bit on the sluggish side. The obvious way out of this is an *awk* compiler, preferably generating something like C as output.

However, on closer inspection, it's not quite so simple. Generating C for *awk* is a straightforward exercise, given an *awk* parser. Unfortunately, the generated C is a mass of function calls. Essentially

all the data operations remain more or less interpretive, done by run-time library functions, with only the flow of control truly compiled. This is a worthwhile speedup, but not entirely satisfactory. To remove *awk* from its current status as a second-class citizen, what is wanted is an *optimizing* compiler.

For example, there is no inherent reason why an *awk* variable used only as a counter should not be compiled into a C integer, so that statements like

```
for (i = 1; i < NF; i++)
```

would run at essentially the same speed as if they were written that way in C. (In the general case there would be a slight added overhead, because integer overflow would have to be caught and referred to a more general version of the code, but most *awk* implementations limit the maximum value of *NF* to the point where even that would be unnecessary.) Naive code generation for this, however, spends vast amounts of time checking to be certain that *i* is never a string, which can usually be established by inspection of the program at compile time.

For another example, there is no need to do dynamic space allocation for the result of, say

```
s = substr($0, 1, 5)
```

since it is known to be at most 5 characters long. Space allocation is a prominent feature in the run-time profiles of *awk* programs that do a lot of string manipulation. Not *all* of it can be eliminated, but with careful data-flow analysis, a worthwhile fraction could be.

On a broader scale, *awk* programs that are written using the pattern+action scheme can spend a lot of time repeatedly checking input lines for various conditions. Often the time needed for this could be greatly cut down if the patterns were compiled together, rather than as completely independent entities. As a gross example from *awf*, if the pattern

```
/^\.(ta|ll|in|ti|po|ne|sp|pl|nr)/
```

is not matched, there is no need to even consider the later pattern

```
/^\.(sp)/
```

(The reason for this slightly odd-looking arrangement is that the first pattern picks out requests that need to have an arithmetic expression processed before they are executed.) More mundanely, an input line that fails to match `/^\.(ne)/` because its second character is not "n" need not even be tried against `/^\.(nr)/` later.

Even more broadly, multi-pass *awk* programs often are clearer and simpler than a single monolith that does the same job. However, they suffer from the high overhead of I/O on their connecting data streams. Often there is no fundamental obstacle to compiling them into a single C program, eliminating

the overhead entirely, by correlating output from one pass with input to the next and making the link directly.

There are other useful tools that are not merely too slow, but completely missing as a result of *awk*'s heritage as a "glue" language. The one that almost any *awk* programmer wishes for, usually very quickly, is an *awk* debugger. As it is, *awk* debugging is back in the dark ages of inserting `print` statements and staring at the code.

Another missing tool is an *awk* profiler. This is particularly galling given the poor performance of current *awk* implementations, since there is great incentive to tune for speed and no good way of doing so. At present, the only feasible tuning technique is to rely on intuition—a notoriously unreliable guide in this area—to identify bottlenecks, and then do before-and-after timings to try to decide whether a possible improvement really helped. The unsatisfactory nature of this procedure helps to explain why a lot of *awk* programs are slow.

The various more minor tools for *awk* programming—customized editing facilities, libraries of useful functions, cross-referencers, etc.—are also worthy of note, but many of them would start to evolve fairly naturally if the bigger problems were solved and *awk* became a credible language for major programs.

The obvious question at this point is whether existing tools could be adapted to solve some of the big problems. Unfortunately, the situation doesn't look promising.

The hard part of the optimizing *awk* compiler is its optimization, not the mundane issues of parsing etc. Although concepts can be borrowed from existing work on data-flow analysis and the like, much of the implementation seems specialized enough that it would probably have to be done from scratch (unless, perhaps, an optimizing compiler for a similar language were available as a starting point).

Existing multi-language debuggers would provide at least a minimal debugging facility for compiled *awk* programs, but there might be difficulties with data representations and the presentation of source code, especially given serious attempts at optimization. Also, debugging is one area where a suitably instrumented interactive interpreter is generally superior. Given how slowly UNIX acquired such a tool even for C, *awk* programmers probably should not hold their breaths. More modest tools like customization for existing multi-language debuggers and tracing options for existing interpretive implementations would be easier.

It is *almost* possible to profile *awk* programs using the existing UNIX profiling facilities. Of course, one can do profiling, but it tells one much more about the *awk* interpreter than about the *awk* program in question, and data about the former's

execution is only occasionally informative about the latter. The problem is that the existing facilities profile based on the hardware's program counter, not the equivalent in the *awk* interpreter. This could be dealt with by extending the UNIX profiling facility very slightly, so that assignment of profiling "ticks" to bins could be done based on the value of a programmer-supplied variable rather than the program counter.

Alternatives

Another possibility for dealing with *awk*'s problems is not to fix *awk*, but rather to attempt to identify its best features and transplant them to another language, the obvious candidates being C and C++. The potential of this approach is limited, since the concise notation will be lost to some degree. However, better subroutine libraries for C and class libraries for C++ would be substantial improvements in those languages too [1], so this is worth pursuing even if *awk* does become a credible tool for large jobs. Libraries for dynamically-allocated strings (including input and output), field structuring of input, regular expressions, and associative arrays could make a wide variety of C/C++ programs more robust and easier to write and read.

The other alternative solution is simply to use a different language, such as ICON [7]. This is potentially a satisfactory solution for a single site, although in general *awk*'s competitors are less concise for simple problems. The major difficulty with this approach is portability. *Awk* at least is widespread within the world of UNIX and UNIX-like systems, and there is hope that *nawk* may achieve similar status eventually. In terms of availability over a large number of sites and variety of machines, the only competitor for old and new *awk* is *perl*, a much uglier language (it has been described as "*awk* with skin cancer") with similar performance problems.

Conclusions

Awk is really a much-underestimated language. Contrary to popular belief, using it for large programs is quite feasible. The programs are a fraction of the size of C code, much easier to write and modify, and much easier to verify against specs.

UNIX support for *awk* is poor, however, most especially in the lack of a compiler. Compiling *awk* well appears to be possible, although doing good optimization is tricky. Better tools for debugging are also desirable, and very small changes to existing software would make useful *awk* profiling practical.

Given a good implementation and tool set, *awk* could take its place beside C as the preferred programming language for many Unix applications, to the great benefit of programmers and users.

References

- [1] Henry Spencer, "How To Steal Code—or—Inventing The Wheel Only Once", *Proceedings of the USENIX Technical Conference*, February 1988, pp. 335-345.
- [2] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, "AWK—A Pattern Scanning and Processing Language", in *UNIX Programmer's Manual*, Seventh Edition, Volume 2, Holt, Rinehart, and Winston 1983, pp. 451-459.
- [3] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger, "The AWK Programming Language", Addison-Wesley 1988.
- [4] Geoff Collyer and Henry Spencer, "News Need Not Be Slow", *Proceedings of the USENIX Technical Conference*, January 1987, pp. 181-190.
- [5] Henry Spencer, "nroff -man/-ms clone written in (old) awk", Usenet newsgroup *comp.sources.unix*, vol. 23 issue 27, July 1990.
- [6] R.C. Holt, J.R. Cordy, and D.B. Wortman, "An Introduction to S/SL: Syntax/Semantic Language", *ACM Transactions on Programming Languages and Systems*, Vol 4 No 2, April 1982.

Henry Spencer is head of the Zoology computer facilities at University of Toronto. He was educated at University of Saskatchewan and University of Toronto, both of which may prefer not to admit it. He is the author of several freely-redistributable software packages, notably the original public-domain *getopt*, the redistributable regular-expression library, and the *awf* text formatter. He can be reached as henry@zoo.toronto.edu, or, if you must, as Henry Spencer, Zoology Computer Systems, 25 Harbord St., University of Toronto, Toronto, Ont. M5S 1A1 Canada.



Program Loading in OSF/1

Larry W. Allen, Harinder G. Singh, Kevin G. Wallace,¹
Melanie B. Weaver – Open Software Foundation

ABSTRACT

In recent years, program loading facilities in UNIX have become more advanced. Support for position independent, relocatable shared libraries can be found in several UNIX systems today. For OSF/1, we have designed and implemented a program loader with several important new abilities. In addition to supporting shared libraries, it supports callable loading and unloading of object modules, a flexible symbol resolution policy based on *packages*, and loading and unloading of kernel modules. Moreover, the loader supports object file format independence, through use of a *loader switch*. Here, we present an overview of the OSF/1 program loader, describing its major features and capabilities. We present design decisions and tradeoffs we made in implementing the loader, with special emphasis on key issues such as symbol resolution policy, object file format independence, and the loader support required for loading of kernel modules.

Introduction and Goals

The primary job of a program loader is to load an executable object file into memory and prepare it for execution. In traditional UNIX systems, an executable object file must have no unresolved external references, and must have all its address references relocated to absolute addresses. These restrictions were imposed because the program loader (implemented as part of the *exec()* system call in UNIX) was quite simple. The simplicity led to very fast program loading; however, it also caused the program loading facility to be quite inflexible. For example, traditional UNIX systems did not support any sort of shared library facility; nor was it possible to dynamically load an executable module into a running process without completely replacing the entire address space. As run-time libraries have become larger, and as the ability to dynamically customize both the operating system and applications has become more critical, the traditional restrictions have become intolerable.

We designed and implemented a new program loader to alleviate these restrictions as part of the development effort for OSF/1, the Open Software Foundation's XPG3-conformant, Mach-based operating system. Our primary requirement in designing the OSF/1 program loader was that it must be able to load a program that might contain *unresolved references*, and that might be *relocatable*. A program with unresolved references has some external symbols that have not been assigned addresses when the program is linked together; these references must be *resolved* by the program loader before the program can be run. A relocatable program has not yet

had absolute addresses assigned to its code and data references, and hence can be loaded at any available address in the process address space; however, the program loader must *relocate* it (assigning addresses to all code and data references) before it can be run. The ability to load and execute relocatable programs with unresolved references is the primary requirement for supporting shared libraries and callable program loading.

A number of goals shaped the architecture of the OSF/1 loader. First, and most important, was that most of the loader not depend on the format of executable object files, and that it be easy to add support for new object file formats. Many vendors have a large investment in their own object file formats and in language tools to generate and manipulate them; we wanted to preserve this investment. We also anticipate improvements in object file formats in the future, and we want it to be easy to upgrade the OSF/1 loader when necessary.

We wanted the loader to be very flexible in terms of the kinds of program loading it could support. In addition to supporting the UNIX *exec()* primitive and shared libraries, we also wanted the loader to support a set of callable loader interfaces, allowing a program to explicitly load and unload modules from its address space at run time. Furthermore, we felt it would be possible to use the same loader technology to support loading and unloading modules from the operating system kernel dynamically at run time as well. Other design considerations were:

- We felt that *exec()* performance was crucial. The most frequently used UNIX commands are small and run for a short duration; it is quite important that the loader not impose too high a load-time penalty on these commands.
- We wanted to cleanly separate the *policies* enforced by the loader from the *mechanisms*

¹Author has been on sabbatical at the Open Software Foundation and is an employee of the Hewlett-Packard Company.

implementing those policies, and we wanted the policies to be independent of the object file format in use.

- If possible, we wanted to implement as much of the loader as possible outside of the operating system kernel.

The remainder of this paper explains the design and implementation of the OSF/1 program loader, and how it met these goals. It begins with a description of the overall architecture of the loader itself, followed by sections on the use of the loader in supporting *exec()* and kernel loading. It gives a brief description of the OSF/ROSE object file format, which supports the advanced loader functions described previously. It ends with a discussion of performance and some speculations about future work.

Loader Architecture

Overview of the Loader Architecture

As mentioned above, the primary job of the loader is to load object modules that may contain unresolved references and that may be relocatable. For each object module loaded, this job involves three primary phases:

1. Symbol resolution. This phase determines which other object modules to load to resolve a module's unresolved symbols.
2. Address assignment and region mapping. This phase maps the module's executable code and data into the address space.
3. Relocation. This phase fixes up the relocatable locations in the module to correctly address the text and data addresses (in this module and in other modules) being referenced.

Conceptually, loading is a recursive process - symbol resolution requires loading other modules upon which the module being loaded depends.²

Loader Structure

The OSF/1 program loader is a separate, executable object module, mapped into the address space of every process at a fixed absolute address. The loader runs in user mode, with the same privileges as the main program being run in the process. The loader exports a number of routines that provide the functions of the loader application programming interface. These routines can be referenced by the main program or shared libraries being loaded into the address space.

To meet the goal of object file format-independence, we split the basic loader functions outlined above into format-independent and format-

dependent sections, with well-defined interfaces between the sections. The format-dependent managers provide the minimal routines necessary to access the information in the object file, and to perform work that inherently depends on the object file format, such as relocation. The bulk of the loader is format-independent.

In particular, all of the policy-setting code in the loader is format-independent; this includes the code implementing the symbol resolution policy and the address space allocation policies. By keeping the policy-setting code format-independent, we ensure that the loader *semantics* seen by an application program do not depend on the particular object file format being used. Format-independent policies also make it easier to change the policy, without requiring changes to all the format-dependent managers.

The interface between the format-independent portion of the loader and the format-dependent managers is through a vector of procedures called the *loader switch*. The procedures in the loader switch and the way it is used are described in detail below.

Loader Clients

In the UNIX operating system, execution of a new program begins with a call to *exec()*, which discards the current address space and loads the new program into a clean address space in the current process. In OSF/1, the loader is mapped into the clean address space as part of the *exec()* operation for programs with unresolved references. The OSF/1 *exec()* system call begins in the kernel by loading the program loader into the clean address space and passing control to the loader. The loader in turn loads the requested program and all the other object modules (shared libraries) on which it depends. The loader remains in the process' address space so that it can service calls to load further modules, or to obtain information about what other modules are loaded. The detailed operation of the *exec()* call is described in a later section.

The very same loader that is used for loading modules into user-space processes suffices for loading modules into the kernel as well. For kernel loading, the loader runs as part of a privileged user-space process, the *kernel load server*, which services requests from operator commands to load and unload kernel modules. Kernel loading involves the same steps listed above (symbol resolution, address assignment and region mapping, and relocation), followed by the copying of the relocated text and data into the kernel's address space. A later section describes kernel loading in detail.

In addition to *exec()* and kernel loading, the loader exports a set of functions that can be called by application programs to permit the explicit

²In the actual OSF/1 implementation, the recursion is flattened to iteration.

loading and unloading of object modules, and to support various query operations. This loader application programming interface is described in the next section.

The Loader Application Programming Interface

The loader provides an application programming interface (loader API) for program loading, dynamic loading of modules across processes, and for debugging support. The loader API calls are implemented in the loader and exported to client applications.

Callable Load and Unload Interfaces

As mentioned earlier, dynamic loading and unloading facility is provided to the application via the callable *load* and *unload* interfaces.

The *load* system call permits a running process to cause modules to be loaded into its virtual address space. The loaded module can have imports that resolve either to other modules, to shared libraries that have previously been loaded, or to other shared libraries (see "Symbol resolution policy") whose modules will then be loaded. On success, the *load()* system call returns an identifier known as a module ID. Module IDs provide a convenient way to reference loaded modules from other loader related functions.

The *unload* system call causes a module that has previously been loaded to have its image unloaded from the process address space. There is no attempt to unsnap any links. This implies that any references to an unloaded module will remain as dangling references. It is the responsibility of the application to make sure that it does not reference unloaded modules. No reference counting mechanism was built into the OSF/1 loader to deal with callable unload. It was deemed complicated and expensive, and fails to account for pointer references.

Other useful calls include *ldr_entry()*, which returns the entry point (if any) of a loaded module, and *ldr_lookup_package()*, which looks up the address of a symbol within a module by package name (see Packages in "Symbol Resolution Policy").

Cross-process Loading

The loader API contains a set of interfaces that permit a process to invoke loader operations on other processes. For completeness, we have defined these cross-process operations (and the debug support operations in the next section) generically to work on any process. However, the cross-process operations are only used for dynamically loading modules into the kernel. The debug operations are supported across arbitrary processes for debugging.

A process can load modules into and unload modules from another process, given sufficient privilege. All cross-process interfaces take a process identifier as an argument.

Before a process can perform any loader operation on another process, it must first "attach" to the other process using *ldr_xattach()*. This performs any necessary initialization, including setting up communication channels between the two processes. The *ldr_xdetach()* call terminates cross-process operations by destroying communication channels set up by *ldr_xattach*.

Once the *ldr_xattach()* function has been performed, the cross-process operations *ldr_xload()*, *ldr_xunload()*, *ldr_xlookup_package()* and *ldr_xentry()* are entirely similar to their counterparts in the single process case. The only difference is the extra argument to specify the process on which the operation is being performed.

Information and Debug Support

The loader provides interfaces for fetching information about loaded modules. These are intended for use by a debugger. Like the calls above, these are also cross-process calls, because they are designed for use by a debugging process. Similarly, they all take a process identifier as an input argument to uniquely identify the process about which information is to be provided.

The *ldr_next_module()* call is an iterator that provides to the caller the next module ID of a loaded module in the target process' address space. This module ID can then be used to obtain other specific module information such as the pathname of the module, the number of regions in the module, and so forth, through the use of the *ldr_inq_module()* call. The *ldr_inq_region()* call is used to obtain specific information on each region of a loaded module. Such region-specific information includes the virtual address of the start of the region, its size, protection, and other information.

Installation of Shared Libraries

The *ldr_install()* and *ldr_remove()* calls provide the loader API interfaces for installing and removing shared libraries from a process' private list of installed libraries. A process can invoke *ldr_install()* to install the specified shared library in its private list of installed libraries. This list is inherited by child processes on *fork()* and is retained across *exec()*. The *ldr_remove()* call removes a shared library that was installed in the current process' private list (see next section).

Symbol Resolution Policy

The Symbol Resolution Problem

Symbol resolution is the operation of determining an absolute value for each unresolved symbol in an object module. When loading an object module that has unresolved ("import") symbols, the loader must determine an absolute value for each import symbol before the program can be executed. The symbol values are used in the relocation phase of loading to patch the instructions and data of the module being loaded to the external locations being referenced.

The loader must resolve each import symbol in two stages:

1. It must determine the pathname of the object module (shared library) that *exports* the symbol, and arrange to load that object module into the process (if it is not already loaded).
2. It must determine the absolute value of the symbol.

These two stages must be separate, because the symbol value may depend on the address at which the exporting module is loaded.

The primary policy issue to be addressed in symbol resolution, then, is this: Given an unresolved import symbol, determine the pathname of the object module that exports the symbol. The policy chosen will affect the flexibility of the loader and shared library system — the simplicity of installing and maintaining programs and libraries, and the ease with which a user can override standard symbol resolutions. It will also have some effect on load-time performance.

A secondary policy issue is the question of *when* symbol resolution occurs. The loader could resolve an import symbol at any point between the loading of the module importing the symbol and the execution of the first instruction referencing the symbol. The policy chosen will affect the load-time and run-time performance of programs.

Requirements for the Symbol Resolution Policy

We started with several requirements when choosing the symbol resolution policy to be used by the OSF/1 program loader. The first and most important was that, in the default case, the user should not have to be concerned with symbol resolution. This means that any information required by the loader for symbol resolution should normally be supplied by the programmer or the program installer (system administrator) at the time the program is built or installed. We felt it would be unusual for the user running a program to want to control the symbol resolutions. Moreover, we felt strongly that the user should not have to configure his environment correctly (by setting up a set of search rules, for example) to get programs to work correctly.

On the other hand, a second requirement was that it must be *possible* for a user to control symbol resolution for programs without having to modify the programs in any way. This is a requirement for developing and debugging shared libraries, and also for customizing an application by supplying a private version of some library the application uses. User control of symbol resolution is also important in an environment where unprivileged users may install programs that include their own shared libraries or dynamically loaded modules.

We also did not want any symbol resolution information stored in the object file to include pathnames of library files. There is no guarantee that a library file will be installed at the same pathname in the development and target systems. Sharing libraries among multiple systems across a network is complicated if the library file pathnames must be the same on all systems sharing the file. Also, storing full library pathnames in each object file makes it very difficult to move a library in the file system hierarchy — doing so requires finding and modifying each object file that uses the library.

Finally, if possible we wanted to alleviate the problem of name conflicts. The problem is that most UNIX system and library interfaces use short, frequently-used symbols (such as *read* and *time*) as interface names. This causes the serious potential problem that more than one library may attempt to export unrelated interfaces with the same name, leading to ambiguity at symbol resolution time. We saw this problem arising in other UNIX systems supporting shared libraries, and felt that it would become even more of a problem in the future, as application writers begin to take advantage of the shared library facilities.

With respect to the policy issue of *when* symbol resolution should occur, we felt it would be adequate for the OSF/1 loader to resolve all unresolved symbols at program load time. Deferring symbol resolution to later times (such as on the first reference to an unresolved symbol) can reduce the initial time required to load the program, but at a potential cost in run-time performance. Moreover, deferring symbol resolution can lead to the possibility of a program failing after hours or days of execution, when it tries to reference an unresolved symbol for the first time and the loader is unable to resolve the symbol. We will re-examine the issue of deferred symbol resolution once we have some results from performance analysis of the loader.

Symbol Resolution Policy in OSF/1: Packages

The requirements listed above imply the need to store some symbol resolution information in each object module, and also to supply some symbol resolution information to the loader at run time, via some database that is common to all the modules. It

must be possible for a system administrator to manage the common database, and for any user to add to or override the common database when necessary. To this end, in the OSF/1 loader we introduced the concept of a *package*.

Packages in OSF/1 provide a way to specify and control the locations of groups of symbols, without embedding absolute pathnames in the object modules importing those symbols. In the OSF/1 model, a shared library or other object module is viewed as being composed of one or more packages, each of which exports one or more symbols. Each symbol exported by the module belongs to exactly one of the module's packages, and each package in a system resides entirely in one object module. An object module with unresolved import symbols also lists the package to which each import symbol belongs. The loader's job is to match up the import packages required by an object module being loaded with the packages exported by the libraries installed in the system; it uses a set of run-time tables for this matching.

For package matching to work, each package name must be unique across the set of all packages exported by all the libraries installed in the system. A symbol name, however, only needs to be unique within a given package: two packages can each export a symbol named "*fork*" without conflicting. This alleviates the problem with symbol name conflicts described above.

The following sections discuss packages in more detail.

Binding Symbols to Packages

The programmer has a great deal of flexibility in grouping exported symbols into packages. We felt that, in general, a package should be a semantically related grouping of interfaces, much like a "module" in the Modula-2 language or a "package" in Ada (the inspiration for the name). However, we were unwilling to require language support for OSF/1 packages, due to the need to support the very large existing base of C programs. So, OSF/1 implements the symbol-to-package binding, for both import and export symbols, at *link* time, through information supplied to the UNIX link editor program, *ld*.

The OSF/1 linker and the OSF/ROSE object file format allow the programmer to restrict the set of external symbols that are exported and visible from an object module. When linking a module that is to export some symbols (such as a shared library), the programmer must specify the symbols to be exported as part of the linker command. This specification includes the name of the package to which each exported symbol belongs. The symbol-to-package-name association is recorded in the executable object module. This information is used by the loader at library installation and load time.

The symbol-to-package-name association is also used by the linker, when linking other modules that will import symbols from this module. A module with unresolved import symbols must be linked against the shared libraries (or other object modules) exporting the unresolved symbols. Unlike linking against a static library, linking against a shared library does not result in the linker combining the modules and assigning a value to each import symbol; instead, the linker simply verifies that every import symbol is exported by one of the libraries, and determines the name of the package to which the symbol belongs. This import package name is stored with the import symbol for use by the loader during symbol resolution.

In the future, we hope to extend this method for managing package names to allow for source-level specification of package names, both in languages that already support a package concept (such as Modula-2 and Ada), and in C. It ought to be possible to add a pragma or other extension to the C language to permit the programmer to group the exported symbols into packages in the source declarations. This would have the advantage of keeping all the interface declaration information for all the symbols of a package together in the source code, and would simplify use of the linker.

Package Name to Library Name Translation

When the loader is requested to load a module with unresolved import symbols, it must translate the import package names obtained from the object file into the pathnames of the libraries or other modules that must be loaded to resolve the imports. It does this translation by looking up each package name in a set of run-time tables, the *known package tables*, to obtain a library pathname.

To implement the symbol resolution policy, the OSF/1 loader supports a three-level hierarchy of known package tables. The base of the hierarchy is the *Global Known Package Table*. This is a single, system-wide table, shared by all users of the system, that lists the default locations for all of the generally-available packages. The Global Known Package Table is maintained by a privileged system administrator. The system administrator uses the *lib_admin* command to install a list of libraries into the Global Known Package Table, normally at system boot time; these libraries are then available for symbol resolution by any program run on the system.

To allow individual users to override the global translations, and also to permit users to install their own private translations, each process in the system may also have a *Private Known Package Table*. The Private Known Package Table is a rather unusual data structure, in that it is inherited from a parent to child process across a UNIX *fork()* operation, and retained in the process' address space

across an `exec()` operation. This unusual semantic (similar to the behavior of environment variables) allows a user to override a package name translation (or install a new, private package name translation) via a shell command. The loader then uses the private translation in loading all the commands the user subsequently executes from that shell. Each of the OSF/1 shells has built-in `inlib` and `rmlib` commands to install and remove libraries from the shell process' Private Known Package Table.

In addition to the Global and Private Known Package Tables, the loader maintains an additional table of package-name-to-module translations for each process: the *Loaded Package Table*. The Loaded Package Table contains translations for all the packages exported by each module that has been *explicitly* loaded into the process (i.e., not implicitly loaded by the loader to resolve an import symbol). The Loaded Package Table allows a main program, for example, to export symbols that can be used to resolve imports from a module loaded by a call to the `load()` system call.

The loader searches the known package tables in the following order:

- Loaded Package Table
- Private Known Package Table
- Global Known Package Table

It is important to note that the user does not need to override an *entire* package when installing an override translation into the Private Known Package Table. When the loader finds a translation in the package tables, it verifies that the module exports the desired symbol within the package. If not, it continues searching the tables for other modules exporting the same package. This allows a user debugging a shared library, for example, to install a test library that only overrides the routines being debugged; programs will resolve the overriding routines in the test library, and all the other routines in the package in the global library.

Future Extensions to Packages

As mentioned above, in the future we would like to allow the programmer to specify the package name to which symbols belong at the source code level, via a C language extension or pragma. We have also discussed adding *version information* to packages. Since a package is a semantically related grouping of interfaces, packages provide the right level of granularity for version checking. Version checking on an entire library is too coarse-grained (a change in an unrelated interface will cause a version mismatch), while version checking on individual interfaces is impractical to specify and manage.

Example of Symbol Resolution Using Packages

This section gives an example of building a shared library and a program to use it on OSF/1, and the steps involved in installing the library and running the program. Imagine that we have a set of three related routines, `init`, `process`, and `term`, which are to be implemented in a shared library. Since the routines are related, it is reasonable to package them together. Therefore, we choose a unique name for the package, `test_package`. The package will be exported by a new shared library, `testlib.so`, and we will write a program that makes use of the new routines from the shared library, `testprog`.

The first step is to write the routines, compile them, and link them together to form the shared library. One of the command line arguments to the linker, when linking together the library, will be the name of the package and the list of all the symbols it contains. The linker command will look roughly like this:

```
ld -o testlib.so testlib.o -export \
    test_package:init,process,term
```

Next, we write the `testprog` program, compile it, and then link it against our new shared library. The linker uses the list of exported symbols and their package names from the shared library to determine the package names for the imported symbols in our main program. The command line to compile and link the test program is, roughly:

```
cc -o testprog testprog.c testlib.so
```

Now, before the command can be executed, the shared library must be installed, so that the loader can translate the import package name it obtains from the test program into the library pathname. In this case, assume that this is a private library; hence, the user installs it into the Private Known Package Table for his shell, using the `inlib` command as follows:

```
inlib testlib.so
```

This causes the loader to read the list of exported packages from the test library, and install the package name to library name translations in the shell's Private Known Package Table. The translations will be inherited by commands run from that shell. Therefore, when the user executes the command

```
testprog
```

from that shell, the loader now has all the information it requires to resolve the imported symbols. In this case, all three imported symbols belong to a single package (`test_package`). The loader looks up the package name in the known package tables in order. It finds the package name translation in the Private Known Package table, and arranges to load the `testlib.so` library into the process. Later in the loading process, the loader obtains the absolute

values of the imported symbols (based on the virtual address at which the library is loaded), and uses these values to relocate the references to the symbols from the main program.

Comparison With Previous Policies

Several previous implementations of shared library loaders for versions of the UNIX operating system have used other policies for symbol resolution. This section describes some previous approaches and compares them with packages.

AIX Version 3.1

AIX Version 3.1 from IBM includes a program loader that provides a very similar set of functions to those provided by the OSF/1 program loader.[3] The AIX program loader, however, uses a combination of search rules and a library leaf name stored in the object module for symbol resolution. Each imported symbol specifies the leaf name of the library from which the symbol is to be resolved. The loader searches for a library with this name in a list of directories specified by a search rule, which is obtained either from the object module or from an environment variable set by the user.

We felt there were a couple of disadvantages to the AIX V3.1 policy. Obtaining a search rule from an environment variable requires that every user must ensure that the environment variable be set correctly in order to run any application that uses shared libraries, a difficulty for naive users. AIX V3.1 permits a default set of search rules to be stored in the object module. However, this policy lacks flexibility, particularly in a network, where a library may reside at different pathnames on different machines. Search rules can also lead to problems with accidental overriding of symbol name translations if a new file with a leaf name that duplicates an existing library is created in a directory earlier in the search path. Also, we felt that binding a symbol name to a library leaf name was unnecessarily restrictive, in that it makes it difficult to move an interface from one library to another. AIX provides a "forwarding" mechanism that alleviates this problem somewhat.

System V Release 4.0

System V Release 4.0 uses a symbol resolution policy that is similar to the AIX V3.1 policy described above. However, an ELF object module (the object file format used in System V Release 4.0) does not list the library leaf name to be used for resolving each symbol; instead, it simply lists the leaf names of all the libraries on which the module depends.[2] The order in which the libraries are listed is critical, because symbol resolution is done by a breadth-first search of the library dependencies in the order listed. The System V Release 4.0 loader searches for the library dependency directories using a search path either from the object file or

from an environment variable, like the AIX V3.1 loader.

The System V Release 4.0 loader does not implement an explicit *load()* system call. This is partly because it is difficult to specify the effects of the call on the symbol resolution policy. Because the order of loaded modules is significant for symbol resolution, the caller would have to specify the proper place in the list of loaded modules to insert the newly loaded module to get the desired symbol resolution behavior. Moreover, the System V Release 4.0 policy does nothing to alleviate the problems of name conflicts among libraries. The advantage of the System V Release 4.0 scheme is that it maintains the exact symbol resolution semantics of the traditional UNIX link editor.

Domain/OS

The Domain/OS operating system, from the Apollo Computer Division of Hewlett-Packard, provides a program loader that uses a very different symbol resolution policy.[5] In many ways, it was the inspiration for the OSF/1 policy. In Domain/OS, the program loader performs symbol resolution by looking up each unresolved symbol in a set of tables of installed libraries. The installed library tables list, for each symbol, the complete pathname of the library to be loaded to resolve that symbol. The loader looks up each symbol first in a per-process installed library table (inherited across *fork()* and maintained across *exec()*), and then in a global installed library table.

The OSF/1 symbol resolution policy is quite similar in essence to the Domain/OS policy. The biggest difficulty we saw with the Domain/OS policy was the problem of symbol name conflicts: in Domain/OS, all symbols exported by all shared libraries must be unique across the entire set of installed libraries. We felt that this restriction would become intolerable as the use of shared libraries expanded. We were also concerned about the growth in the size of the installed library tables. Packages alleviate both of these problems.

The Loader Switch

As described above, the OSF/1 loader was designed to support multiple object file formats. For each of these object file formats a different (but functionally similar) set of routines acts upon the object files. This set of routines comprises a format-dependent manager. The format-independent routines (called the *loader*) perform operations common to all object file formats. A *loader switch* provides the interface between the loader and the various format-dependent managers. In this section, we describe the procedural interfaces of the loader switch that form the boundary between the loader and the format-dependent managers.

Switch

Each loader switch entry consists of a group of procedure pointers which represent a single format-dependent manager.

<i>lsw_recog</i>
<i>lsw_get_imports</i>
<i>lsw_map_regions</i>
<i>lsw_get_export_pkgs</i>
<i>lsw_lookup_export</i>
<i>lsw_relocate</i>
<i>lsw_get_entry_pt</i>
<i>lsw_run_inits</i>
<i>lsw_cleanup</i>
<i>lsw_unload</i>

Loader Switch Entry

Key loader switch operations

The loader invokes the procedures of the loader switch in four distinct phases. These are:

1. recognition
2. symbol resolution
3. address assignment and region mapping
4. relocation.

In the first phase, the recognizer routine (*lsw_recog()*) takes a file descriptor and examines the file to determine whether its object format is the kind handled by this format-dependent manager. The loader walks the list of loader switch entry structures invoking each format-dependent manager's recognizer in turn, until one of them recognizes the file's object format. The loader then uses this format-dependent manager for its actions on this file. However, if the list of format-dependent managers is exhausted, and the file has not been successfully recognized, the loader attempts to dynamically load new format-dependent managers to attempt recognition of the file format (see "Dynamically Loaded Format-Dependent Managers"). If all known format-dependent managers fail to recognize this file, the loader returns ENOEXEC and fails to load the file.

Once the recognizer routine has successfully determined that the object file can be handled by this format-dependent manager, it returns a format-dependent handle to the loader. The loader uses this handle and this format-dependent manager for all future actions on this object file.

In the second phase of loading, the loader attempts symbol resolution. The loader gets a list of the object module's imports (package and symbol names) through *lsw_get_imports()*. The function *lsw_get_export_pkgs()* returns a list of the packages exported by a module. The loader uses

lsw_lookup_export() routine to look up values of exported symbols. Using these routines, the loader is able to resolve the imports of this module with exports of previously loaded modules or installed libraries. If the object module being loaded has imports that are not resolved, the loader fails to load it.

The third phase assigns addresses to various regions and maps regions from the object module. A *region* is a virtually contiguous piece of a process' address space. All the executable code and data for a module are located in its regions. The regions of a module are loaded through the *lsw_map_regions()* routine.

In the final phase, once all regions for a module are loaded and its imports resolved, the *lsw_relocate()* function is called to relocate all relocatable addresses throughout the various regions of the module.

The *lsw_cleanup()* routine is invoked by the loader at the end of successful loading of the module. This allows the format-dependent loader to clean up any of its data structures that are no longer needed and close any files that it may have open. The procedure *lsw_get_entry_pt()* provides the entry point (if any) of the loaded module. Prior to the execution of the module's entry point, the module initialization routines are executed via *lsw_run_inits()*.

The format-dependent module unloading operations are performed by the *lsw_unload()* routine. It unmaps the various regions associated with the module and destroys any remaining format-dependent data structures.

Dynamically Loaded Format-Dependent Managers

Format dependent managers can be statically built-in to the loader or added dynamically. There are several advantages to this:

- The size of the loader can be kept small by statically linking in only frequently used managers.
- It is unnecessary to rebuild the loader every time a new format-dependent manager is to be added.

The loader starts with a set of statically bound format-dependent managers. It only attempts to load a new format-dependent manager when none of the existing managers recognize a file. To do this, it reads an ASCII file containing a list of dynamically loadable format-dependent managers. Each format-dependent manager has an entry point that it uses to install itself at the end of the loader's list of managers. The loader loads the next previously unloaded manager and attempts recognition of the object file. If this new format-dependent manager fails to recognize the object file, the loader will try

again. Only when all the secondary format-dependent managers have been loaded, and the file format cannot be recognized by any of them, does the loader fail to recognize the object file.

Overview of a Format-Dependent Manager

Each format-dependent manager consists of an entry point and a set of routines corresponding to the entries in a loader switch structure. As format-dependent managers themselves depend entirely on the object file format they support, it is difficult to discuss their internal structure in a general way. We will look at two data structures that are common to most format-dependent managers. These are the format-dependent handle and the export table.

The handle of a module is a format-dependent data structure. It is returned to the loader by the recognizer. Because the handle is passed to all other loader switch procedures, it is a good place for the format-dependent loader to save module specific data it needs across loader switch calls. Such information includes pointers to object file headers, other object file meta-data, pointers to data structures used by the format-dependent manager (e.g., export symbol table), and the file descriptor associated with the module. The contents of the handle are private to the format-dependent manager and vary vastly between different managers.

The export symbol table (commonly a hash table for efficiency) is maintained by the format-dependent loader. This contains the list of symbols exported by this module and their values. It is used for looking up symbol values (*lsw_lookup_export()*) for the loader. This data structure is completely private to the format-dependent manager. The *lsw_lookup_export()* procedure is different from many of the other loader switch interfaces (e.g., *lsw_get_imports()* or *lsw_get_export_pkgs()*) in that it performs the lookup operation itself, rather than returning a pointer to a data structure that it has built for the loader.

exec() Architecture

In OSF/1, the kernel is able to load programs that are absolute and have no unresolved external references. However, for programs that are relocatable or have unresolved external references, OSF/1 employs the loader to load the program, relocate it and resolve its unresolved external references. Note that this implies that the loader itself must be absolute and may not have any unresolved external references.

We have extended the architecture of *exec()* to accommodate the loader and we have kept knowledge of the loader outside of the program being loaded. This yields a more flexible implementation and separates mechanism from policy. Other systems embed knowledge of the loader and even

selection of the loader into the program itself. In OSF/1, the program has no knowledge of whether a loader is needed to load it or whether the kernel can load it. If a loader is needed, the program has no knowledge of which loader will be used. For example, the OSF/1 loader is capable of loading absolute programs, a task typically relegated to the kernel.

exec_with_loader()

We call our extension to the *exec()* architecture *exec_with_loader()*. *exec_with_loader()* functions similarly to *execve()*, except that rather than loading the program, it loads a loader instead and simply passes the name of the program to the loader. The expectation is that the loader will then eventually load the program. *exec_with_loader()* manipulates the address space, file descriptors, signal state, IPC state, *close-on-exec* processing, and so forth just as *execve()* does.

exec_with_loader() is a system call that programs can explicitly call as well as an internal kernel function that *execve()* can call implicitly. The system call interface to *exec_with_loader()* is as follows:

```
extern int
exec_with_loader (
    int flags,
    const char *loader,
    const char *file,
    char * const argv[ ],
    char * const envp[ ] );
```

The *loader* argument to the *exec_with_loader()* system call allows the caller to specify the loader. The *loader* specified may be NULL, in which case *exec_with_loader()* selects the system default loader (*/sbin/loader*). The *file* argument points to the name of the program to be loaded and the *argv* and *envp* arguments are the same as they are to *execve()*.

exec_with_loader() honors the *set-user-ID* and *set-group-ID* mode bits of the program file. For security, *exec_with_loader()* forces all programs that are to be run *set-user-ID* or *set-group-ID* to be loaded by the system default loader regardless of whether a valid *loader* is passed to the *exec_with_loader()* system call, under the assumption that the system default loader is secure. Note that the settings of the mode bits on the loader file have no effect.

exec_with_loader() handles *#!* interpretation of program files just as *execve()* does. That is, *exec_with_loader()* reformats the argument list and causes an interpreter to be loaded rather than the program file. Note that the loader file cannot be subject to *#!* interpretation.

Programs rarely call *exec_with_loader()* explicitly as a system call. Instead, *exec_with_loader()* is typically called implicitly from *execve()*. For

example, when the name of a program is given to a shell, the shell calls `fork()` and `execve()` in order to run the program. One of the first things that `exec()` must do before loading the program is to decide whether it *can* load the program. For example, the program may be in an object file format not understood by `execve()`. More typically in OSF/1, the program may be relocatable or have unresolved external references. In other words the program may require shared libraries. In OSF/1, `execve()` itself cannot load such programs. Instead, `execve()` calls `exec_with_loader()` to load the system default loader and have the loader load the program.

To make this decision, `execve()` abstractly employs a list of multiway recognizers, one for each object file format it supports. Each recognizer inspects the program file and makes one of the following decisions about it:

- Accept the program file. `execve()` can load it and does so.
- The recognizer recognizes the program file, but `execve()` cannot load the program; for example, because the program is relocatable or has unresolved external references. However, the recognizer knows that the loader can load the program. Therefore the recognizer arranges to call `exec_with_loader()` to load a loader to load the program.
- Reject the program file; that is, the file is not in the object file format recognized by this recognizer. In this case, pass the file onto the next recognizer. If all the recognizers reject the file, `execve()` returns `ENOEXEC`.

Once the multiway recognizers collectively make a decision, `execve()` takes the appropriate action.

Kernel Communication with the Loader

After `exec_with_loader()` loads a loader, it informs the loader about the program file that is to be loaded by passing the name of the program file to the loader. As with System V Release 4.0, we use the *auxiliary vector*[9] for communication between the kernel and the loader. We have defined three new types of auxiliary vector entries for this communication.

AT_EXEC_FILENAME

This entry contains a pointer to the program filename passed to `execve()` or `exec_with_loader()`.

AT_EXEC_LOADER_FILENAME

This entry is optional and contains a pointer to the loader filename.

AT_EXEC_LOADER_FLAGS

This entry is optional and contains one-bit flags intended for use by the loader.

The kernel passes the name of the program the loader is to load in the `AT_EXEC_FILENAME` entry. Note that all programs receive an auxiliary vector, even ones directly loaded by `execve()` rather than by a loader. Such auxiliary vectors typically contain only one entry, one of type `AT_EXEC_FILENAME`. The filename associated with this entry can be used to reliably and unambiguously determine the filename of the program loaded, rather than relying on the caller of `exec()` to correctly set `argv[0]`.

Under certain circumstances, the loader may need the name of its own program file. The `AT_EXEC_LOADER_FILENAME` entry contains that name.

The kernel passes various flags to the loader in the `AT_EXEC_LOADER_FLAGS` auxiliary vector entry. These flags are a combination of the flags from the *flags* argument to the `exec_with_loader()` system call and system flags defined by and set by the kernel. Currently defined are flags to indicate whether the process is running with *set-user-ID* or *set-group-ID* or whether the process is being traced (*a la ptrace()*). Trusted or secure implementations of the loader can use the *set-user-ID* or *set-group-ID* flags to determine if certain actions should be taken to avoid compromising system security; for example, ignoring any inherited known package tables. The loader uses the trace flag to determine if it should communicate with the debugger to signal the completion of key loader events, such as when loading of the main program and its dependencies has been completed.

In OSF/1, programs reference the auxiliary vector in one of two ways, either as the fourth argument to `main()` or through the external variable `_auxv`.

Program Launch

After the kernel loads the loader and passes it the auxiliary vector, the kernel transfers control to the loader's entry point in the loader's `crt0` routine. `crt0` eventually calls the loader's `main()` function. To a large extent, `main()` functions like any other application that calls the loader application programming interfaces. `main()` fetches the name of the program to load from the auxiliary vector. It then calls `load()` to load the program and its dependencies. Next, it calls `ldr_entry()` to get the entry point of the program. `main()` returns the value of the program's entry point to `crt0`. `crt0` cuts back any local storage remaining on the stack and transfers control to the program's entry point. Note that the loader remains in the process address space, so that it can respond to future loader requests.

Lessons Learned and Planned Improvements

In the short time we have worked with the `exec()` architecture, we have learned a few lessons and we plan to make some improvements in a future

release of OSF/1.

Originally we had wanted shells to directly call the loader to load and run new programs. This would obviate the need for calling `exec()` in the kernel and totally eliminate that overhead. The loader would prevalidate the program being loaded as much as possible. For example, the loader would potentially handle `#!` interpretation. Unfortunately, this scheme really could not work. The semantics of `exec()` dictate much that either must be done in the kernel or would be difficult at best to do in the loader, such as manipulation of file descriptors, the signal state and `set-user-ID` processing. Since OSF/1 still supports absolute programs that have no unresolved external references, we wanted the efficiency of having the kernel load those programs rather than relying on the loader. In the end we found our original scheme to be unworkable and abandoned it.

We had hoped to rewrite `exec()`, but all that we had time to do was to extend the `exec()` architecture to accommodate `exec_with_loader()`. We still do intend to rewrite `exec()` for a future release. We plan to make it more object oriented and base it on an `exec switch`, similar in concept to the loader switch.

The OSF/1 loader, like other user-space loaders, suffers from the lack of an atomic commit. In other words, if the loader fails to load a program, it is impossible to return an error from the original call to `execve()` because the calling program has already been overlayed. We had briefly considered performing the load operation in another address space and atomically substituting the new address space for the old one, once the load operation succeeded. We believe the overhead and complexity of this approach with respect to our implementation to be too expensive.

In a future release we plan to pass to the loader a file descriptor on the program file to load, in an `AT_EXEC_FD` auxiliary vector entry, rather than passing the filename of the program to load. This should give us a performance improvement and also allow us to support execute-only access for programs loaded by the loader.

Kernel Loading

OSF/1 supports the dynamic loading of object modules into the kernel. This facility is general purpose in nature and is generally not found in other UNIX systems. OSF/1 typically uses this facility to dynamically load subsystems such as file systems, device drivers, network protocols and STREAMS modules into the kernel. A configuration of OSF/1 need only load those subsystems that will actually be used and thus unused subsystems do not use kernel resources, such as memory.

We discuss kernel loading in this paper because it is an interesting application of the OSF/1 loader. With the addition of a few relatively minor ancillary services, the OSF/1 user-space loader loads modules into the kernel. These services include:

- A process or server, called the *kernel load server*, to maintain kernel loading state information.
- An interprocess communication (IPC) or remote procedure call (RPC) facility for communication between the server and any of its clients.
- Creation and maintenance of the initial kernel export list.
- Support for kernel address space manipulation.

A Simple Kernel Load Request

All kernel load requests are directed to the kernel load server. Conceptually, there are several steps that take place when a client communicates with the kernel load server. First a client sends a message to the kernel load server, requesting a service, for example, to load a file. The kernel load server receives the message and calls the loader to load the file using a *loader context*³ specific to the kernel. This kernel context is different from the normal process context in that it specifies the kernel versions of the region allocation functions that actually allocate address space within the kernel. Note that the kernel load server creates and maintains this kernel context.

During loading the loader calls a format-dependent map region routine to map all the regions in the file. When the loader maps a region of a file, it maintains both a *virtual address* and a *map address* for the region. For normal process loading, these addresses are the same. However, for kernel loading, the *virtual address* is the address at which the region is to be mapped in the kernel address space, while the *map address* is the address at which the region is to be mapped into the address space of the kernel load server. The loader maps and relocates the kernel module in the kernel load server's address space. The entire essence of our scheme to load a module into the kernel is that when the loader does relocation processing, it patches the module with addresses from the kernel address space (i.e., *virtual addresses*) rather than addresses from the address space of the kernel load server (i.e., *map addresses*). The format-dependent map region routine and the kernel versions of the region allocation

³A loader context is a closure that contains the loader state for a given *process*. This state includes a list of modules loaded into the context, a module name hash table, descriptors for the known package tables and the region allocation and deallocation functions for the context.

functions work in conjunction to choose where the regions eventually reside when copied into the kernel, and where they will live while locally mapped into the kernel load server's address space. The kernel versions of the region allocation functions by necessity make a system call to actually allocate space within the kernel.

After mapping the regions into the kernel load server's address space, the loader relocates them and then returns the module ID of the newly loaded module. From the loader's perspective, the load operation is complete. The kernel load server iterates over all regions associated with the module just loaded, inquiring to get the *virtual address, map address, size and protection* of each region. The kernel load server then makes system calls to actually load (e.g., map, copy, etc.) the regions of the newly loaded module from the kernel load server's address space into the kernel's address space. Lastly, the kernel load server sends a reply message back to the client, returning any return values, such as the module ID of the module just loaded or an error code upon failure.

Upon completion of the kernel load request, a client typically calls the kernel load server to get the module's entry point, and then the client typically calls the kernel module's entry point in the kernel to have the kernel module initialize itself, for example, by plugging itself into the appropriate switch table.

Programming Interface

The programming interfaces for kernel loading are the loader cross-process interfaces: *ldr_xattach()*, *ldr_xload()*, *ldr_xentry()*, etc. The following simple code fragment illustrates how an application could dynamically load the NFS-compatible kernel module and get the module's entry point for initialization. The example ignores any error returns.

```
#define KMOD "/sbin/subsys/nfs_kmod"

{
    ldr_process_t  kernel;
    ldr_module_t   module_id;
    ldr_entry_pt_t entry;

    kernel = ldr_kernel_process();
    ldr_xattach(kernel);
    ldr_xload(kernel, KMOD,
              LDR_NOFLAGS,
              &module_id);
    ldr_xentry(kernel, module_id,
              &entry);
}
```

The *ldr_kernel_process()* function returns the loader process identifier that effectively specifies the kernel and the kernel context. The *ldr_xattach()* function performs any necessary cross-process initialization: for example, setting up the necessary communication channels for communication with the kernel load server when called with the value returned by

ldr_kernel_process(). The *ldr_xload()* function loads a file and returns the module ID of the associated module. Lastly, the *ldr_xentry()* function fetches the address of the module's entry point.

Kernel Load Server

The kernel load server is the glue that binds together the loader and the other necessary functionality to provide the kernel loading service. The kernel load server is a privileged user-mode process. It maintains the state information (e.g., modules, export lists, etc.) of what has been loaded into the kernel. It is privileged because it can manipulate the kernel's address space. Communication with the kernel load server takes place via BSD sockets.

The function of the kernel load server is to receive kernel load requests, process them and send back appropriate replies. This is the basic kernel load server loop. Before entering the loop, during server initialization, the kernel load server builds the kernel context and the initial export list of the kernel. The kernel load server is typically started early, during system initialization, by *init*.

Initial Kernel Export List

The kernel load server constructs the initial kernel export list by reading the kernel object file (by default, */vmunix*). To fetch the initial kernel export list, the kernel load server implements its own format-dependent managers. These format-dependent managers differ from the normal ones used by the loader in that rather than being able to load a file, these format-dependent managers can only grab the export list from a file. Thus, to get the initial kernel export list, the kernel load server simply *loads* the kernel object file using one of its own format-dependent managers.

Kernel Address Space Manipulation

There are several requirements for the kernel address space manipulation necessary for the support of kernel dynamic loading by the kernel load server in OSF/1. They include:

- Able to allocate and deallocate kernel address space.
- Callable from user-mode.
- Supports arbitrary combinations of protection (read, write and/or execute).
- Supports wired and paged memory.
- Able to allocate at a fixed address or *anywhere*.
- Functionality available only to privileged processes.

The OSF/1 virtual memory (VM) calls satisfy most of these requirements but there is no call to control the wiring or locking of pages into physical memory. Because of this and due to a lack of time and

resources, we implemented an interim system call, *kloadcall()*, rather than using the OSF/1 VM calls directly from usermode. *kloadcall()* provides direct access to all the internal kernel functions for the manipulation of the kernel address space as needed for kernel loading. *kloadcall()* performs various operations on the kernel address space. We structured the interface to each operation to exactly match its VM call counterpart.

Future Work

In some future release we will complete the work necessary to allow kernel loading to use the OSF/1 VM calls directly from user-mode. We also plan to allow replacement of the use of BSD sockets with the OSF Distributed Computing Environment (DCE) RPC facility for the communication between the kernel load server and its clients. We would also like to experiment with ways of fetching the initial kernel export list directly from the kernel loaded into memory.

Introduction to OSF/ROSE

This section discusses why we designed a new object file format and briefly describes some loader-related aspects of OSF/ROSE. The detailed design of the object format is not discussed because it involves many issues and tradeoffs which are beyond the scope of this paper.

Why a New Object Format?

The loader is designed to be relatively object format independent in order to allow vendors to continue using their own compiler tools or to adopt new ones. However, it cannot achieve full functionality with adequate performance without a certain amount of support from the object file format. We wanted our OSF/1 reference ports to fully utilize the loader, so we needed a suitable object format. Note that we designed the loader first, after becoming aware of the features from many formats, but without being committed to any one format. It turned out that no single existing format had everything that we needed. Therefore, we elected to adapt one of the formats to meet our needs.

Goals for the Object File Format

The three main goals we had for the object format were:

- To support the loading of shared libraries with reasonable performance.
- To be portable.
- To be extensible.

Shared libraries require symbol resolution and relocation at load or run time, as opposed to (static) link time, when performance is more critical. Portability was also important — we needed to minimize

the amount of machine-dependent code in the loader. Finally, knowing that it would be impossible to specify everything that anyone would need in the next few years, we wanted to be sure that there were well-defined ways to add functionality or to adapt to different machines or systems. These three goals conflict with each other, so the challenge was to provide for them all in a balanced way.

Overview

We decided to base our object format on Carnegie-Mellon University's Mach-O [10], which provided the kind of generality that we were seeking. This format is organized as a header, followed by a variable length list of variable length load commands, followed by the program data and *metadata* sections in no particular order. Some of the load commands serve as section headers. Each load command has a type (that defines its structure) and a size.

We found that we were not able to maintain compatibility with the original format and still meet our goals. On one hand, some of the load commands were inappropriate or inadequate. On the other hand, we needed some new load commands to improve support for our shared library loading. Therefore, we changed all the structures and types, inventing a new set of load commands and modifying the load command header. The result was a modified Mach-O format which we have called OSF/ROSE.

We made several structural modifications to the format as well. One major modification was to introduce another level of indirection by adding a "map" of the load commands, thereby allowing metadata in one section to refer to metadata in another section without depending on either file offsets or fixed positions of section headers. Another major modification was to eliminate the multiple file and program section types and to use simpler, more flexible ways of adapting to different situations. Of course, we had to change the symbol information quite a bit, adding package names and enhancing address information.

Almost as important as what we included in the format is what we omitted. In particular, by not specifying how PIC (Position Independent Code) was to be implemented, and by specifying relocation that was flexible and adaptable, we increased the machine-independence of the loader's OSF/ROSE format-dependent manager. The rest of the loader is independent of the specific implementation of PIC. One benefit of his approach is that PIC is needed only for performance, and not for functionality.

Data Representation Issues

The way an object file's metadata is represented has several effects. We considered these issues:

- The ability to have cross-tools.
- The ability to validate files before loading.
- Ease and efficiency of processing.
- Extensibility.

We decided to use a universal *canonical* representation for the header and *native* representation (that used by the local compilers) for the rest of the metadata. Since the header identifies the data representation used in the rest of the file, it is possible to determine unambiguously the data representation of any OSF/ROSE object file. However, because the header is the only structure that has a canonical representation, the loader does not have to spend time translating the rest of the metadata into a representation that it can read.

Future Enhancements

Here are some of the areas that we think should be addressed in the future.

Version Support

When libraries are linked separately from programs, there is the possibility that the calling sequences and data structures used to communicate may become mismatched. We are considering extending the format to support the package version information described above.

Hash Tables for Exported Symbols

The exported symbols do not yet have a hash table because we felt that it was premature at this point to specify one. For now, the loader constructs an export hash table at run time and we can experiment with the effect of alternative approaches on performance.

Performance

In thinking about how to characterize the performance of the OSF/1 loader, we identified two large classes of program:

1. Programs whose total execution times are dominated by "start-up transients", such as the time to load the program and to page-fault in the program's code and data. A remarkably large number of frequently used UNIX commands fall into this category[4]. For these programs, we expected the use of the user-space loader and shared libraries to impose a noticeable performance degradation, and we were quite concerned with characterizing and minimizing this degradation.

2. Relatively long-running programs, whose execution times are not dominated by startup transients. For these programs, we expected negligible performance degradation due to use of the user-space loader and shared libraries (largely due to the extra indirections in position-independent code and data references). We also hoped to see a counteracting performance *improvement* in heavily loaded systems running large numbers of this class of program, because of the reduction in working set due to shared libraries. Unfortunately, we have not yet had time to evaluate the performance of this class of applications.

Initial measurements of *fork/exec/exit* benchmarks (which are a worst-case example of programs dominated by startup transients) have shown more performance degradation when using the user-space loader and shared libraries than we had expected. Our preliminary analysis, using kernel profiling and page-fault statistics, shows that nearly all of the extra time is spent in the kernel in page-fault handling. According to the kernel profiling data, this is mostly due to increased numbers of zero-fill faults (which occur on the first reference to an uninitialized data page) and page reclaims (due to the virtual memory system deferring the installation of a page translation for a resident page into the translation hardware until the first reference to the page).

The increase in the number of zero-fill faults appears to be mostly due to faults on the areas used for dynamic memory allocation of loader data structures. The loader uses a variant of the 4.4BSD *malloc()* algorithm for its dynamic memory management; this algorithm uses a separate page of virtual memory for each power of 2 increase in the size of allocated objects. The result is that an application like the loader, which allocates relatively few objects of many different sizes, requires a very large number of pages, each of which is sparsely used. We are currently planning to prototype a version of the loader memory allocator using a circular first-fit algorithm, which ought to be better suited to the loader's pattern of dynamic memory usage.

The increase in the number of reclaim faults is largely attributable to reclaim faults on the loader code, and to a lesser extent to extra reclaim faults on the shared C library. Reclaim faults on the loader occur because the loader is currently unmapped from the address space in the early phases of *exec()*, and mapped in again later in the same *exec()* call, resulting in all its pages being removed from the address translation hardware. Extra reclaim faults on the shared C library are due primarily to reduced locality of reference.

Our current plans are to begin prototyping some modifications to the *exec()* path to attempt to reduce the number of reclaim faults. Our current idea is to retain the loader code in the process address space

across *exec()* whenever possible. We also want to look at possible tools for reordering the object files in the shared C library to improve the locality of reference.

Of course, once the first line of performance problems have been alleviated, we fully expect to have to do more cycles of profiling, analysis, prototyping, and code modifications.

Conclusions

The program loader described in this paper, including the support for shared libraries, kernel loading, symbol resolution based on packages, and the OSF/ROSE object file format, is currently in use in the OSF/1 operating system on three different machine architectures. In general, we believe it has met the goals outlined in the paper. We are currently investigating potential improvements in loader performance, and are also considering some important loader extensions, such as maintenance of package version information.

References

- [1] James Q. Arnold, "Shared Libraries on UNIX System V," *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA, USENIX Association (1986).
- [2] James Q. Arnold, "ELF: An Object File to Mitigate Mischievous Misoneism," *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, USENIX Association (1990).
- [3] M. A. Auslander, "Managing Programs and Libraries in AIX Version 3 for RISC System/6000 Processor," *IBM Journal of Research and Development*, Vol. 34, No. 1, pp. 1-136 (January 1990).
- [4] Luis-Felipe Cabrera, "The Influence of Workload on Load Balancing Strategies," *Proceedings of the Summer 1986 USENIX Conference*, Atlanta, GA, USENIX Association (1986).
- [5] *Domain/OS Programming Environment Reference* No. 011010-A00, Apollo Computer Inc. (July 1988).
- [6] R. A. Gingell, M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS," *Proceedings of the Summer 1987 USENIX Conference*, Phoenix, AZ, USENIX Association (1987).
- [7] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press (1972).
- [8] Marc Sabatella, "Issues In Shared Libraries Design," *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, CA, USENIX Association (1990).
- [9] *System V Application Binary Interface: WE 32000 Processor Supplement*, Prentice Hall (1990).
- [10] Avadis Tevanian, "Architecture-Independent

Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach," CMU-CS-88-106, Carnegie Mellon University (December 1987).

Larry W. Allen is a member of the technical staff at the Open Software Foundation. He received the Bachelor of Science degree in Computer Science from the Massachusetts Institute of Technology in 1985. From 1985 to 1988 he worked on the Domain/OS operating system at Apollo Computer. Larry joined OSF in January of 1989. He is an architect of the OSF/1 operating system and the project leader for the OSF/1 loader project. His address is: Open Software Foundation; 11 Cambridge Center; Cambridge, MA 01242. Reach him electronically at lwa@osf.osf.org.

Harminder G. Singh is a member of the technical staff at the Open Software Foundation. He is a graduate of the Indian Institute of Technology, Delhi, India. He received his Master of Science Degree in Computer Science from Clemson University in 1987. From 1987 to 1989 he worked on the development of the QIX distributed operating system at Cogent Research Inc, Beaverton, OR. Harminder joined OSF in July 1989. He has been a member of the OSF/1 loader project, and is currently working on the OSF Distributed Computing Environment. He is a member of the Association for Computing Machinery. His address is: Open Software Foundation; 11 Cambridge Center; Cambridge, MA 01242. Reach him electronically at hermi@osf.osf.org.



Kevin G. Wallace has been a member of the technical staff at Hewlett-Packard since 1982. He has worked on the design and implementation of the HP-UX kernel on Precision Architecture. He managed an HP-UX kernel development team, and helped coordinate Hewlett-Packard's OSF efforts. Over the past year he has been on sabbatical at the Open Software Foundation, working as member of the OSF/1 loader project. He received Bachelor of Science and Master of Science degrees from the Massachusetts Institute of Technology in 1982. He is a member of the ACM and the IEEE Computer Society. His address is: Hewlett-Packard Company; HP-UX Kernel Laboratory; Open Systems Software Division; 19447 Pruneridge Avenue; Cupertino, CA 95014. Reach him electronically at kwallace@hpda.hp.com.



Melanie B. Weaver is a member of the technical staff at the Open Software Foundation. She holds a B.A. in Mathematics from Vassar College and worked for many years on the development of the Multics system for Honeywell Information Systems (now Bull), including the dynamic linker. Melanie joined OSF in April 1989. She is a member of the OSF/1 loader project, and was a principal in the design of the OSF/ROSE object file format. She is a member of the ACM. Her address is: Open Software Foundation; 11 Cambridge Center; Cambridge, MA 01242. Reach her electronically at melanie@osf.osf.org.



Compiling from Saved State: Fast Incremental Compilation with Traditional UNIX Compilers

Alastair Fyfe, Ivan Soleimanipour, Vijay Tatkar – Sun Microsystems

ABSTRACT

We describe a strategy for extending traditional UNIX compilers that enables them to be used in new ways, notably as adjuncts of interactive tools such as a debugger and a function-level recompilation tool. The compilers' translation of source to intermediate representation is reused without significant change and two new capabilities are added to each compiler: *saved-state* capability and *compile-server* capability.

We use the term *saved-state compiler* to refer to a compiler capable of storing the context information built up during compilation and later retrieving the stored data, reconstructing the earlier context, and performing further compilation. This capability allows a compiler to quickly reestablish a particular compilation context without needing to reprocess declarations in the program's source.

The context information required for compilation is saved in a special-purpose Symbol Information Database (SID) stored in ELF files and retrieved by the compiler on demand. SID information is useful to tools other than the compiler, and thus may also be accessed by debuggers, performance analyzers and other tools.

We use the term *compile server* to describe a compiler which supports interfaces better suited to interprocess communication than traditional UNIX command-line invocation. By adapting a compiler to support RPC and shared memory, it is possible to minimize the overhead of using the compiler as an adjunct of another tool, such as a debugger.

In this paper we describe the steps involved in adding *saved-state* and *compile-server* capabilities to two production-quality compilers and a preprocessor, discuss three applications of this technology, and gives measurements based on the current implementation.

Introduction

The compiler is the preeminent program development tool, yet the use of traditional UNIX compilers has typically been confined to the program building step that precedes execution. By extending a compiler to save and later restore its compilation context it is possible to also use it for interactive compilation during the debugging and bug fixing that accompany program development.

A compilation context exists inside a running compiler as a collection of interlinked data structures. The crux of this paper is the premise that it is possible to store and later retrieve a context by making moderate changes to the compiler, and that saving a context and accessing it can both be done efficiently. The validity of the premise depends largely on the availability of a suitable database. In the next three sections, we describe requirements for such a database and the particular solution that evolved during this project.

The following two sections describe the work involved in adding saved-state and compile-server capabilities to two compilers and a preprocessor.

There are four key steps in adding saved-state functionality: characterizing the compilation context; identifying data structures that represent the context; modifying the compiler to emit this data in the course of ordinary, build-time, compilation; and modifying the compiler to divert its symbol table lookups to the saved state when invoked for incremental compilation.

Adding compile-server functionality entails repackaging the compiler's translation service so it is available as a low-overhead interprocess procedure call, with source input, error messages and intermediate representation duly handled as input or output arguments.

The remaining sections describe three applications of saved-state technology: (1) pre-compiled header files, (2) compiling an individual function in context, and (3) compiling an individual statement or expression in context. Measurements obtained on real-life source files confirm that compiler access to saved context is fast enough to allow the use of a compiler for interactive translation of expressions and statements during debugging. In general, the cost of faulting in a compilation context saved in

SID is much smaller than the cost of reconstructing it from source declarations.

We conclude with a review of related work and a description of anticipated future directions.

Another Symbol Information Database (SID)

Our analysis of the data management functionality necessary to save and restore a compilation context led to the following major requirements:

- The ability to store and retrieve any data structure that can be represented in C.
- The ability to relocate pointers, so that references from one data structure to another remain valid in different address spaces.
- The ability to compress data when it is stored and expand it when it is retrieved. That is, the ability to maintain distinct on-disk and in-memory representations of each data structure stored.
- Support for concurrent reading of the saved data by more than one process so that, for example, the compiler and debugger may concurrently traverse the same linked list.
- Incremental access to the saved information. That is, retrieval of an individual data structure should bring as little additional data into the address space of the reader as possible.

Both storage and retrieval costs need to be kept small so as not to degrade ordinary compilation during program building and allow quick incremental compilation at some later time.

We assumed the standard UNIX program-building conventions based on make and relocatable and executable files could not be seriously perturbed. This assumption severely constrained possible solutions, limiting them to what could be accomplished by annotating or augmenting ELF files [ATT90].

A review of existing debugger-information formats, including dwarf and stabs [Lint90] yielded no technology that addressed the above requirements. We devised a solution based on three, new, three-letter acronyms: IDX, SID and the PCT.

The Interprocess Data Exchange (IDX) library provides an I/O mechanism for storing arbitrary collections of interlinked data structures in files whose format conforms with the UNIX System V Release 4 executable format (ELF).

The SID layer defines a collection of data structures that are stored with IDX. Different components, such as the compiler, debugger, pre-processor, optimizer, and performance analyzer, can each define private data structures. Other, public, data structures are shared among all components.

One of these public data structures, called a *container*, is noteworthy since it is used to provide access to the other data. Conceptually, information

about a program is organized as a tree whose nodes are *containers* with all available information attached as attributes of some node. Nodes at the top layers of this tree correspond to ELF executable and relocatable files and nodes at lower levels correspond to language-specific constructs such as file, function, and block scopes.

The program container tree (PCT) provides a simple hierarchical model which ensures connectivity and addressability of all the data defined in SID for a given program.

Much of the information we store about programs is inherently hierarchical, so the PCT schema is often a useful one. However it is important to stress that the PCT in no way precludes expression of more complex, nonhierarchical relationships among other data structures stored in SID.

IDX

The purpose of the IDX library is to provide a standard mechanism for passing data structures among processes whose lifetimes do not overlap. Data to be shared is saved in intermediate files which comply with the ELF format.

A simple way to introduce the IDX interface is by analogy with a well-established library which performs a similar service, XDR[Sun90].

XDR is concerned with exchanging data across a network connection whose end points may or may not abide by the same data-representation conventions. The XDR library provides built-in functions for "serializing" scalar data types and makes the user responsible for supplying "filter" functions which "serialize" constructed types, such as C structures. These filter functions are called implicitly as a consequence of network transactions.

Similarly, to use IDX, the user must first register with the library a description of each type of data structure to be stored or updated, and for each such type, provide an appropriate filter function. The filter function is called implicitly each time data is stored (packed) or retrieved (unpacked).

As with XDR, any C data structure can be stored. However references from one data structure to another must be expressed via an opaque, library-supplied type, `IdxHandle_t`, rather than via ordinary pointers. Handles are address-space-independent pointers. Their values are relocated by the library as needed. They are dereferenced by calling a library function, `idx_access()`.

Whereas an XDR filter must distinguish between, say, an int and a float, an IDX filter need only distinguish data that needs be relocated (handles) from data that doesn't.

The key elements of the IDX interface are the following types and operations:

IdxSect_t - An IDX section. A section is an aggregate of data structures which are stored as a group. Typically, all data produced in one compilation are stored in a single IDX section. Each IDX section is stored as multiple ELF sections.

IdxHandle_t - A typeless, address-space-independent, pointer used to express a link between two data structures. The two data structures may, but need not, reside in the same IDX section.

IdxSpace_t - An IDX space is the collection of sections which contain cross-referenced data. That is, if an item in section A references an item in section B, the two sections must be in the same IDX space. Conceptually, an IDX space is an address space in which handle values are addresses. Typically a SID reader or writer is concerned with only a single **IdxSpace_t**.

idx_access() - This function is used to dereference a handle. It returns an untyped pointer whose value is legal in the caller's process. If the referenced item is not in memory, this function is responsible for faulting it in.

idx_alloc() - This function is used to add data to a section. The caller specifies the type of the structure being added and receives storage appropriate for a structure of that type.

idx_sadd() - This function adds a section to an IDX space. Typically, it is used once to specify the initial section of interest. Additional sections are added as a transparent side-effect of **idx_access()** calls.

idx_close() - This function closes an **IdxSpace_t** by writing and compressing all new or modified data to disk. The user-supplied filter functions are called to pack data as a consequence of this call.

A simplified example of a toy data structure and its packer and unpacker is shown below.

```
struct item {
    char *name;
    IdxHandle_t link;
}

pack_item(IdxCopyState_t *statep,
          struct item **itemref)
{ struct item *item = *itemref;
  short len =
    strlen(item->name) + 1;

  idx_xfr_data(statep, &len,
               sizeof(len));
  idx_xfr_data(statep, item->name,
               len);
  idx_xfr_handle(statep,
                 &item->link, 1);
}

unpack_item(IdxCopyState_t *statep,
```

```
          struct item **itemref)
{ struct item *item;
  short len;

  item = (struct item*)
    malloc(sizeof (struct item));
  idx_xfr_data(statep, &len,
               sizeof(len));
  item->name = (char*)
    malloc(len);
  idx_xfr_data(statep,
               item->name, len);
  idx_xfr_handle(statep,
                 &item->link, 1);
  *itemref = item;
}
```

The **pack_item()** function stores a structure of type "struct item" by obtaining the length of the string and then storing the length of the string, the string itself and the handle. The **unpack_item()** function unpacks a structure of the same type by reversing these steps, allocating storage as needed. In a real application, direct use of **malloc()** would probably be too expensive.

The tedium of writing packer and unpacker filters for each data structure can be reduced slightly by combining them into a single function which tests **IdxCopyState_t** to see if a pack or unpack is in progress.

The sequence of key library calls made by an IDX writer is

```
writer() {
    /* initialize an IdxSpace_t */
    IdxSpace_t *bp =
        idx_space_init();
    /* create a section */
    IdxSect_t *sectp =
        idx_screate(bp);
    /* allocate a new record */
    struct item *item = (struct item*)
        idx_alloc(sdp, ITEM_TYPE);

    item->name = "some name";
    item->link = some_handle;
    idx_close(bp)
}
```

The calls for a reader are similar, except that the **idx_alloc()** is replaced by an **idx_access()**.

Space and time performance of the IDX library are important. The library strives to impose minimal space overhead on the user data. Currently, total size is about 30% larger than the user data and we expect to reduce this to no more than 20%.

The library also allows the user considerable flexibility in selecting the appropriate space/time tradeoff. Filter functions for complex data structures can elect to work arbitrarily hard to compress and expand data. In addition, the library provides a facility whereby the user can designate multiple buffer classes and control what buffer class a given record type is stored in. This facility is used to obtain cross-record compression in addition to the intra-record compression provided by packers.

SID

The IDX library provides a general-purpose access method which we use to store and retrieve data. SID is a layer built on IDX which defines the schema of the data to be stored.

SID's definitions are grouped into separate components. Our assumption is that over time the number of SID readers and writers and the total number of type definitions stored in SID will grow, whereas most readers and writers will be interested in a small subset of the available types. Examples of available components include :

sid_common.h - Type definitions which are common to all components. These include the *SidContainer_t* type described above, structures for defining address expressions in the compiled file, and a section header.

sid_acomp.h - Type definitions which hold the compilation context for an ANSI C compiler. These include enough information about variable types, scopes, storage class and storage allocation to reconstruct compiler symbol table entries.

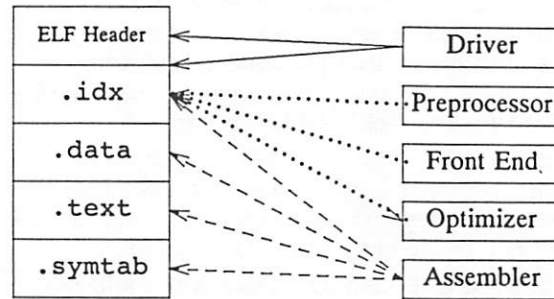
sid_acpp.h - Definitions which hold the compilation context for the preprocessor used with the ANSI C compiler. The key structure is the one used to describe a macro.

sid_debug.h - Definition of data structures relevant to a debugger. These include structures to correlate a source file coordinate with a text location and an inverted index that associates an identifier with its uses.

sid_driver.h - Definitions of structures which record compilation flags.

Components for other languages, or other tools, such as the optimizer or performance analyzer can be added easily. In practice we've observed that, even though IDX makes it trivial to add new data definitions or revise existing ones, co-ordinating a coherent schema for a tools database is difficult.

Integration of SID generation with compilation is illustrated in Fig. 1. The ELF file which will hold the results of compilation is created by the compile driver, rather than by the assembler as is customary. The driver first opens an IDX section within the ELF file and records compile-time flags and component versions. The preprocessor, compiler front end and assembler, successively reopen the IDX section and store new data or update existing data. The assembler also augments the ELF file by adding the conventional products of compilation: the *.text* and *.data* sections and the ELF symbol table. The user can elect to keep all the results of compilation in a single ELF file, or to keep SID in a separate file. Neither approach seems clearly superior to the other.



Driver opens ELF Header and IDX sections.
Preprocessor, Front-End, Assembler
and Optimizer update SID.
Assembler adds *.data*, *.text* and
.symtab sections

Figure 1: SID Generation Across Compilation Phases

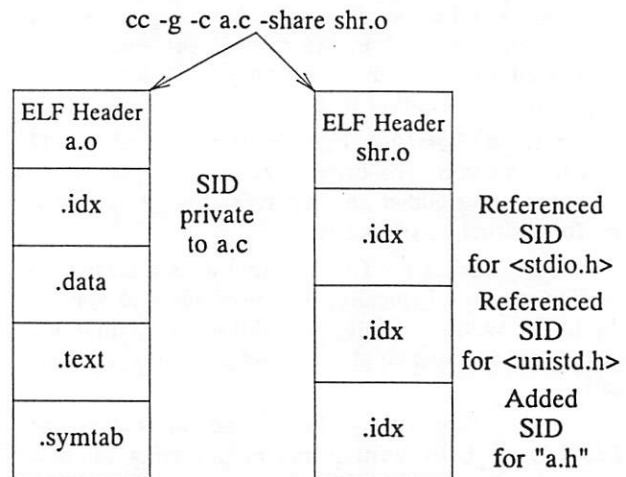


Figure 2: Sharing SID Across Compilations

Sharing SID Across Compilations

To further improve the performance of SID generation, we distinguish data that is private to each compilation from data that may be shared. A significant proportion of the compilation context of many source files is derived from imported interfaces, in C, from text obtained from *#include* files. Typically this information is constant across all files in a program that import those interfaces.

To exploit this fact, the user can elect to separate sharable and non-sharable information. When this is done, the compiler checks whether information has already been produced for a sharable unit and, if so, simply incorporates a reference to that section, rather than regenerating it. Sharable information is stored in a separate ELF file that is referenced by all compilations that contribute to it.

Figure 2 shows an example of this separation. The file `a.c` includes three header files, `<stdio.h>`, `<unistd.h>`, and `"a.h"`. Before `a.c` is compiled, SID information for `<stdio.h>` and `<unistd.h>` is already available in `shr.o`. Compiling `a.c` stores `a.c`'s private SID in `a.o`, references the existing SID for `<stdio.h>` and `<unistd.h>` in `shr.o` and adds SID for `"a.h"` to `shr.o`.

Adding Saved-State Capability

We now outline the steps involved in adding saved-state capability to an existing compiler. The changes required for saved-state support are very specific to each language and compiler. However, the following steps provide a general roadmap for the work to be done.

- Characterize the compiler's context at different points during compilation.

This amounts to asking, "if the compiler were stopped at this point, what information would be needed to later restart it with a new input stream?" The context always includes the compiler symbol table, and usually includes sundry internal variables, such as counters for labels and storage allocation. It may or may not include the state of the parse stack or lexical analyzer.

The goal of this step is to identify a sequence of times at which the compilation context is relatively small. For example, for C and FORTRAN, interesting points occur after all declarations in a given scope have been processed but before the first executable statement in that scope is scanned.

Selecting the granularity of contexts is fairly open-ended. Each language scope must clearly introduce a new context, but it is also possible to introduce contexts for smaller syntactic units such as statements. Ultimately each saved context costs some storage, even if it inherits most of its

information from others, so there is little incentive to introduce more contexts than are needed to support the envisioned applications.

- Define data structures which describe each compilation context.

This step entails separating information which can be reconstructed from information which cannot. For the latter, data structures which can be registered with the IDX library must be invented.

There is a tradeoff between selecting data structures which are close to those that already occur in the compiler and data structures which are designed for general consumption. For example, the representation of language types can either copy the compiler's internal representation or use an alternative representation. On the assumption that the compiler is the most important consumer of this data, we have found it preferable to use data structures which are close to the compiler's existing ones. Readers other than the compiler can transform these representations as needed.

- Modify the compiler to emit descriptions of each compilation context during ordinary compilation.

This is a time consuming but comparatively easy step. The work is not significantly different from what already occurs in the traditional `stab.c` and `dwarf.c` files; however, since more detailed information is recorded, the `sidgen.c` file tends to be somewhat larger. SID generation may be controlled by the conventional `-g` flag.

- Modify the compiler to read and use a previously saved context.

The crux of the work required for this step is to modify the compiler's symbol-table lookup to fault-in previously stored information and to convert between the stored representation and the compiler's internal representation. The compiler obtains a pointer, or more exactly, an `IdxHandle_t` to a saved context and arranges to reestablish its internal state and to then accept

Table 2: File Compilation with Pre-compiled Include Files

File	Conventional Compilation	SID generating	SID reading	File Size (words)	#include Size (words)
tar.c	7.5	11.9 +58%	6.9 -0.8%	4545	1150 20%
regex.c	6.7	10.0 +49%	4.8 -28%	4735	1433 23%
rcp.c	5.4	8.5 +57%	4.2 -22%	2863	943 24%
clntcmd.c	3.7	7.1 +92%	2.4 -35%	1359	2795 67%
status.c	11.9	19.0 +59%	3.4 -71%	2202	30857 93%

new input. For each context, some of the saved information needs to be loaded immediately, but most of it can be loaded on demand. The initial handle to each context is either obtained externally or calculated.

To test these changes we have found it useful to add a special test mode to the compiler named after the `-nodecl` flag that controls it. In `nodecl` mode, the compiler ignores all declarations it encounters in the source and obtains the necessary information from a previously saved context. By compiling standard compiler validation suites first with the `-g` flag and later with the `-nodecl` flag and verifying that both versions execute correctly we obtain a very thorough test of both the SID generation and SID reading components. When compiling a file in `nodecl` mode the compiler must successively calculate the pointer for each context it encounters while processing the file.

Adding Compile-Server Capability

The external interfaces used by traditional UNIX compilers are simple: source and compilation flags provide the input, and the output consists of some intermediate representation, assembler directives, and possibly some error messages. All the complexity of translation remains internal to the compiler.

Though the interfaces are simple, they are oriented to command-line invocation from a shell rather than towards programmable IPC mechanisms. Using the compiler as an adjunct of another tool involves repackaging these interfaces to make inter-process exchange more efficient. Here are the steps we found worthwhile to minimize the overhead associated with invoking the compiler from another tool.

- Devise a simple client-server protocol between the compiler and its client.

The key motivation here is to eliminate the cost of restarting the compiler for each translation request. RPC provides a simple, low-overhead, technique for structuring an inter-process exchange. Though we did not expect to have the compiler and its caller execute on separate machines, we found RPC useful as a tool for structuring the compiler/caller exchange by casting the compiler as a conventional RPC service.

- Provide compiler reset functionality.

Batch compilers tend to assume a separate process will be used for every file compiled. Invalidating this assumption required adding code to reset globals and static variables, plug memory leaks and distinguish per-process initialization from per-context initialization.

- Arrange to pass compiler output via shared memory.

Intermediate files are the conventional medium

used for passing compiler output. To avoid the delay in writing to the file system, we found it worthwhile to exchange the results via shared memory. To lessen the impact on the compiler, we use an allocator which doles memory out of a shared region. Data to be written to the intermediate file is allocated with this special-purpose allocator and thus is instantly visible to the client at end of processing.

- Eliminate assembler directives.

The three applications described below use compile-server and saved-state capabilities in different ways. For pre-compiled header files and single-function recompilation, the results of the compiler are passed to a code generator and assembler in the usual way to obtain a relocatable file. For expression evaluation in the debugger, the intermediate representation obtained from translation is passed to the debugger where it is interpreted in the context of the process being debugged. For this last application, we found it worthwhile to replace the assembler directives output by the compiler with a binary data representation.

For example, when translating the fragment

```
printf("hello world");
```

the compiler expresses the call to `printf` in the intermediate representation and the constant "hello world" via assembler directives. Rather than add a scanner for assembler syntax to the debugger, we found it easier to adapt the compiler to use a simple binary interface called SOD (Structure of Data) to pass the data directives that result from translation.

Table 1: ANSI-C Compiler Changes

Functionality	Lines in Modified Files	Lines in New Files
SID generation	66	1984
SID reading	604	1170
Resettability and RPC	668	382
Shared Memory and SOD	196	0

Table 1 summarizes the costs of adding saved-state and compile-server capability to an ANSI C compiler whose total size is about 61738 lines. The total changes involve around 5000 lines of code. By comparison, approximately 6000 lines are devoted to expression evaluation.

The deliverables for this work consist of two executables, both built with the existing compiler `Makefile` and sources. The first of these is a modified version of the conventional compiler. The `-g` flag controls SID generation and the `-nodecl` flag controls the SID-reading test mode described earlier. The second executable is a version of the compiler

that operates as an RPC service and can pass the results of compilation either via the usual intermediate files or via shared memory. It would be possible to eliminate the first executable at the cost of rewriting the compiler driver.

Pre-Compiled Header Files

We now consider three applications of this technology - what new uses of a compiler are possible once it supports compile-server and saved-state capability? The three applications examined are: pre-compiled header files, compiling an individual function in context, and compiling an individual statement or expression in context.

Whereas all three rely on the ability to compile from saved state, only the last depends on compile-server functionality. In discussing these applications our intent is to quantify the behavior of the modified compiler, not to describe each application in full.

In many large-scale applications the cost of compiling an individual compilation unit is dominated by the cost of compiling the interfaces imported into it. Techniques for compiling interface modules into representations which support efficient import have been studied for some time [Gutk86, Fost86].

The results reported in Table 2 were obtained by compiling five files¹ in each of three modes: conventional compilation, conventional compilation augmented by generation of compiler context information, and compilation using saved context. In the last mode, all the text obtained from header (`#include`) files was deleted. Since the compiler relied on saved context rather than on declarations in the source to obtain its compilation environment, this deletion did not affect the results of the compilation.

The purpose of these measurements is to estimate how much conventional compilation can be speeded up by using pre-compiled symbol table structures rather than text to represent the definitions in standard interface files. The potential improvement for certain kinds of files is considerable: `status.c`, an Xview application which imports 93% of its text from header files benefits from more than a three-fold improvement in compile time when saved state, rather than text, is used to hold `#include` file definitions.

Thus `status.c`, which consists of 2202 tokens² and includes 30857 more during pre-processing

¹The five file considered were obtained as follows: `rcp.c` from the 'rcp' program, `regex.c` from the regular-expression parser in GNU Emacs, `status.c` from the Xview version of dbxtool, `clntcmd.c` from the network module of a prototype debugger, and `tar.c` from the 'tar' program.

²The word token is used incorrectly here: we report the wc program's word count for the pre-processor output.

requires 12 seconds³ for a conventional compilation. When generating SID, this time increases to 19 seconds, a 59% increase. When saved state is used to obtain include file contents compilation requires 3.4 seconds, a 71% decrease. All three of these compilations produce the same relocatable file after code-generation and assembly.

Results for the other files show how these timings vary for files where the proportion of included text is not as extreme as for Xview applications.

Two important qualifications accompany the above results. Including a file of text is a low-level, and thus flexible and unstructured, mechanism for importing a public interface. Several issues which are not covered here need to be resolved before the above technique can be used reliably: these include verifying that the included text corresponds to the saved state and deciding where to store the saved state.

Secondly, all timings are for front-end processing only. This is but one of the determinants of overall compilation time. The timings do not incorporate code generation, assembly or linking.

Table 3: Single-Function Recompile (Saved State)

File Size (words)	Compilation Times		
	Smallest	Average	Largest
tar.c	0.7 7	0.9 137	1.8 794
regex.c	0.7 35	1.2 641	2.0 1900
rcp.c	0.7 19	0.9 172	1.8 1008
clntcmd.c	0.7 4	0.9 86	1.2 388
status.c	1.4 8	1.5 294	2.0 1068

Table 4: Single-Function Recompile (Elided Text)

File	Compilation Times		
	Smallest	Average	Largest
tar.c	1.3	1.2	2.2
regex.c	1.0	1.7	2.9
rcp.c	1.2	1.5	2.5
clntcmd.c	2.0	2.1	2.5
status.c	9.5	9.5	10.2

³All timings reported in this paper are the sum of system and user time reported under release 4.1 of SunOS on a Sun 4/65.

Single Function Recompile

Table 3 illustrates another application of adding saved-state capability to a compiler: an individual function is recompiled in the context of the file in which it was originally defined. In collaboration with the debugger and the dynamic linking facility in UNIX System 5 Release 4, the compiler can then be used to support a "fix-and-continue" feature. The motivation for this application is to increase programmer productivity by shortening the duration of each edit-compile-debug cycle.

For each of the five files considered above, we recompiled three functions, the largest, smallest and median, against the saved context obtained from compiling the entire file (see Table 2).

Function-level recompilation can also be obtained by deleting from the file all function definitions other than the one of interest, retaining all other declarations and `#include` files, and using conventional compilation. For comparison, we include compiler timings obtained with both the saved-state and elided source methods in Tables 3 and 4 respectively.

The purpose of the measurements below is to demonstrate that recompilation of an individual function, using context saved in SID is faster than whole-file recompilation. For each of the five files considered, single-function compilation time never exceeds 3 seconds, though whole file recompilation requires between 6 and 12 seconds.

Thus the largest function in `status.c`, which with 1068 tokens represents 3% of the 33059 tokens in the file after preprocessing, requires 2 seconds to recompile with saved state and 10 seconds to recompile using an elided text approach. Overall, the difference between the two approaches seems negligible unless large included interfaces are involved.

Once again, a number of issues other than compilation need to be resolved before this technique can be put into practice. These include loading the recompiled function into the target address space, resolving any collision between the old and new instances of the function and updating the debugger's symbol manager to incorporate SID for the new version of the function.

The strength and weakness of this approach to incremental compilation is that the compiler has no responsibility for understanding and managing change. Components outside the compiler must detect the change, analyze its impact, update program-wide information to retain consistency, and patch the running image. The compiler is simply presented with a compilation context and asked to compile in that context.

Debugger Expression and Statement Translation

Debuggers allow a user to explore and modify a program by evaluating source fragments in the compilation context set by the current execution point. To evaluate source fragments, debuggers such as `dbx`[Lint90] have duplicated parts of the compiler's translation functionality in the form of an expression interpreter.

An alternative approach, illustrated in Fig. 3 is to use the compiler as an adjunct of the tool. For each expression to be evaluated, the debugger calls the appropriate compile server, passing it a handle to a compilation context and a source fragment. The compiler translates the fragment to low-level intermediate representations, IR and SOD, and returns them along with relevant status and error messages. The IR is then interpreted in the debugger, accessing the target process as necessary. SID for the program being debugged is available in the program's object

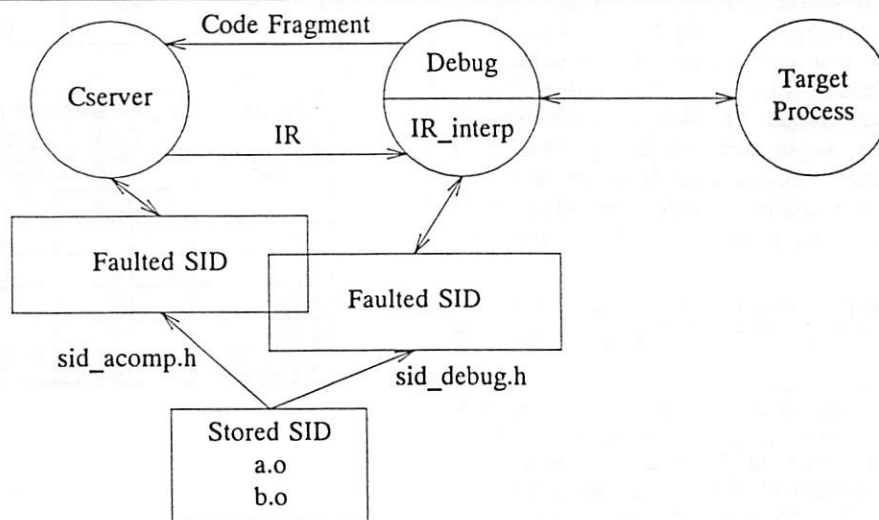


Figure 3: A Compile Server As Translator for Debugger Expressions

files and is faulted in as necessary. The debugger and compile-server are interested in different subsets of the available SID data, and will typically fault in data structures defined in `sid_acomp.h` and `sid_debug.h` respectively.

To facilitate parsing of source fragments in the compiler we've found it useful to embed them into larger syntactic fragments. For example,

```
print p->next
```

might be wrapped as

```
bogus_function() {
    bogus_call(p->next);
}
```

before being transmitted to the compiler.

The advantages of reusing a compiler rather than duplicating parts of it in the debugger are many: complete language coverage, access to thorough test suites via the *nodecl* mode described above, consistency between compile-time and debug-time evaluation, and concentration of language implementation expertise with the compiler staff.

Reuse of the compilers also provides two benefits which are directly visible to the user. Statements can in many cases be translated and interpreted as easily as expressions. Secondly the compiler can be used to construct new language types. Thus the debugger is not limited to descriptions of the types recorded when the program was compiled.

Table 5 gives the timings that result from having a compile server do translation of expressions and statements on behalf of a debugger. This application provided the initial impetus for the development of saved-state and compile-server capabilities. For four different compilation contexts⁴ the table gives the time required to transmit a simple expression to the compile server, have it translated and receive the results of the translation. For the compilation contexts measured, this time does not exceed 1.0 seconds.

Table 5: Debug-Time Expression and Statement Translation			
File	Compilation Times		
	Smallest	Average	Largest
tar.c	0.79	0.77	0.75
regex.c	0.84	0.93	0.80
rcp.c	0.70	0.74	0.74
clntcmd.c	0.86	0.90	0.80

These timings do not include the costs of interpreting the IR relative to the target process after it has been received.

⁴Reliable data was not available for `status.c`.

In the current implementation the compiler reinitializes its internal state at the start of each translation request. This is not required; eliminating it will improve the above numbers.

The cost of initially setting up a compilation context is not included in these numbers. In practice this means that the first expression evaluated in a given context requires about twice as long to evaluate as subsequent expressions.

Selective Loading of Saved State

The importance of loading the contents of a saved context into the compiler selectively, based on demand, has been noted before [Gutk86]. It seems reasonable to assume that only a fraction of the information available in a context is actually required for compilation and therefore that CPU and memory performance can be improved by on-demand loading.

Table 6: Saved State Usage			
File	records/bytes/percentage		
	Generated	Whole file	Single func
tar.c	4493	1013	42
	126365	51050	21230
	100%	40%	17%
regex.c	3721	461	54
	108401	42706	25981
	100%	40%	24%
rcp.c	3135	580	67
	91817	32765	17921
	100 %	36%	20%
clntcmd.c	3097	567	29
	114402	43718	22773
	100%	44%	22%
status.c	6597	611	92
	296721	88617	66956
	100%	30%	23%

The IDX library faults in data from disk as required to satisfy `idx_access()` calls and thus can be easily instrumented to test the above assumption, as shown in Table 6. For each file we show the total number of records saved in the compilation context for the entire file, the number faulted in to recompile the entire file, and the number faulted in to recompile the first function in the file.⁵ The second line in each table entry gives equivalent statistics in terms of the number of bytes read and written, and the third line expresses this as a percentage.

Thus the saved context for `status.c` includes 6597 records, of which only 611 are required to recompile the entire file.

⁵Both recompilations use the SID reading *nodecl* mode described above.

This sort of instrumentation has intriguing possibilities. It could be used to support a winnowing tool which reported to the user the interfaces needlessly imported into a compilation unit. It could also be used to derive the sort of fine grained compilation dependencies discussed in [Tichy86].

Related Work

The most successful incremental development tools, notably those for LISP and Smalltalk[Gold84] have evolved in monolithic, interpreter-based environments. Projects which aimed to support incremental development for conventional languages through compilation technology have relied on a special-purpose compiler tightly integrated into the development tool: Pecan[Reiss83], IDE[Fei82], Integral-C[Ross86], DICE[Fritz83]. There are few descriptions of the successful re-use of existing compilers, loosely coupled to development tools, to support interactive compilation [Pro89, Crowe85]. Techniques for speeding up compilation by using precompiled information are discussed in [Tichy86, Gutk86, Fost86].

Future Directions

The basic directions for future work in this area are adding additional languages and tuning. The techniques described provide a wealth of opportunities for improving space and time performance. None of these has been seriously explored. Data compression and eliminating redundant information stored by separate SID clients will reduce disk space usage. Caching strategies, improved memory management and the addition of index structures to the compiler's saved context will improve performance.

The applicability of these techniques to other languages remains an open question. We expect no major obstacles in adapting a FORTRAN compiler but do not have data for C++.

Summary

We have described a strategy which enables traditional UNIX compilers to be used in new ways. The key elements of this strategy are two additional compiler capabilities, saved state and compile server, and a flexible, ELF-based, program information database (SID). To date our work has been limited to the C language, however the techniques described have no known dependencies on C which would preclude their application to other languages.

By allowing resources already invested in existing compilers to leverage new functionality, this strategy provides a gradual but effective way to evolve a more tightly integrated software development environment.

Acknowledgements

Paul Baclaski participated in the design of SID and implemented early versions of SID generation in the *pcc* compiler and libraries. Brent Benson was responsible for all aspects of saved-state functionality in the ANSI C pre-processor. John Rose contributed important insights to the design of SID and IDX. Robert Corbett and Kathy Stark participated in detailed reviews of the design and provided useful references to related work. Scott Hanham, manager of the first two authors during most of this project, provided encouragement and a good environment in which to do this work.

We thank Nanette Harter, Mark Liu, Steve Muchnick, and Dave Weatherford for reviewing an earlier draft of this paper.

References

- [Crowe85] Crowe, M., Nicol, C., Hughes, M. and Mackay, D. "On Converting a Compiler into an Incremental Compiler", *SIGPLAN Notices*, Vol. 20(10), pp 14-22, October 1985.
- [Tichy86] Tichy, Walter F. "Smart Recompilation", *ACM Transactions on Programming Languages and Systems*, vol. 8(3), pp. 273-294, July 1986.
- [Gutk86] Gutknecht, Jurg, "Separate Compilation in Modula-2: An Approach to Efficient Symbol Files", *IEEE Software*, pp. 29-38, Nov. 1986.
- [Fost86] Foster, David G., "Separate Compilation in a Modula-2 Compiler", *Software Practice and Experience*, vol.16(2), pp. 101-106, February 1986.
- [Lint90] Linton, Mark, "The Evolution of Dbx", *Proceedings USENIX Summer Conference*, pp. 211-220, June 1990.
- [Pro89] PROCASE Corporation, "SMARTsystem Technical Overview", 1989.
- [Fritz83] Fritzson, P. "Symbolic Debugging Through Incremental Compilation in an Integrated Environment", *The Journal of Systems and Software* 3, pp.285-294, 1983.
- [Fritz88] Fritzson, P. "Incremental Symbol Processing", Technical Report LITH-IDA-R-88-09, Department of Computer and Information Sciences, Linkoping University, April 1988.
- [Fei82] Feiler, P. H., "A Language-Oriented Interactive Programming Environment Based on Compilation Technology", PhD Thesis, Carnegie-Mellon University, 1982.
- [Ross86] Ross, G., "Integral C - A Practical Environment for C Programming", *SIGPLAN Notices*, Vol. 22(1), pp. 42-48, December 1986.
- [Reiss83] Reiss, Steven, "PECAN: Program Development Systems That Support Multiple Views", Brown University, Dept. Of Computer Science, Technical Report CS-83-29
- [Gold84] A. Goldberg, *Smalltalk80: The Interactive Programming Environment*, Addison-Wesley,

Reading, Massachusetts, 1984

[ATT90] AT&T Inc., *System V Application Binary Interface*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[Sun90] Sun Microsystems Inc., *Network Programming Guide*, Mountain View, California, 1990

Alastair Fyfe received a B.A Degree in Mathematical Statistics from George Washington University (Washington D.C.) and M.A and M.S. degrees in Statistics and Computer Science, respectively, from the University of California, Berkeley. He has worked with UNIX systems since 1980, primarily in the areas of compilers, optimizers, and network services. None of his code really worked as intended. This led to a fascination with debuggers and more recently to the study of why debuggers don't work as intended. For U.S. Mail, his address is Sun Microsystems, MTV12-40, 2550 Garcia Ave., Mountain View, Ca. 94043. For electronic mail, use alastair@sun.com.

Ivan Soleimanipour is Russian Iranian. He came to America in 1979, and graduated with a BS in computer science from UCSB in 1984. His first job after school was at Culler Scientific working on debuggers. He later did the same at SAXPY Computer and eventually joined Sun Microsystems in late 1988. His current work is *still* on debuggers and his electronic address is ivan@sun.com.

Vijay Tatkar received his B.Tech. in Chemical Engineering at IIT (Madras) in 1983. His quest for better process controllers quickly drew him closer to hardware. The degree bug bit him hard and 3 M.S. degrees later, he realized that the only reality was software. His last M.S. at Rice University in 1988 sealed his career in compilers. He joined Sun Microsystems in 1988. Since then he has worked hard to achieve invisibility, because he believes that the greatest compliment to software, and to compilers specifically, is when users don't notice their existence. Vijay can be reached by electronic mail at tatkar@sun.com.

A New Hashing Package for UNIX

Margo Seltzer – University of California, Berkeley
Ozan Yigit – York University

ABSTRACT

UNIX support of disk oriented hashing was originally provided by *dbm* [ATT79] and subsequently improved upon in *ndbm* [BSD86]. In AT&T System V, in-memory hashed storage and access support was added in the *hsearch* library routines [ATT85]. The result is a system with two incompatible hashing schemes, each with its own set of shortcomings.

This paper presents the design and performance characteristics of a new hashing package providing a superset of the functionality provided by *dbm* and *hsearch*. The new package uses linear hashing to provide efficient support of both memory based and disk based hash tables with performance superior to both *dbm* and *hsearch* under most conditions.

Introduction

Current UNIX systems offer two forms of hashed data access. *Dbm* and its derivatives provide keyed access to disk resident data while *hsearch* provides access for memory resident data. These two access methods are incompatible in that memory resident hash tables may not be stored on disk and disk resident tables cannot be read into memory and accessed using the in-memory routines.

Dbm has several shortcomings. Since data is assumed to be disk resident, each access requires a system call, and almost certainly, a disk operation. For extremely large databases, where caching is unlikely to be effective, this is acceptable, however, when the database is small (i.e. the password file), performance improvements can be obtained through caching pages of the database in memory. In addition, *dbm* cannot store data items whose total key and data size exceed the page size of the hash table. Similarly, if two or more keys produce the same hash value and their total size exceeds the page size, the table cannot store all the colliding keys.

The in-memory *hsearch* routines have different shortcomings. First, the notion of a single hash table is embedded in the interface, preventing an application from accessing multiple tables concurrently. Secondly, the routine to create a hash table requires a parameter which declares the size of the hash table. If this size is set too low, performance degradation or the inability to add items to the table may result. In addition, *hsearch* requires that the application allocate memory for the key and data items. Lastly, the *hsearch* routines provide no interface to store hash tables on disk.

The goal of our work was to design and implement a new package that provides a superset of the functionality of both *dbm* and *hsearch*. The package had to overcome the interface shortcomings cited above and its implementation had to provide performance equal or superior to that of the existing

implementations. In order to provide a compact disk representation, graceful table growth, and expected constant time performance, we selected Litwin's linear hashing algorithm [LAR88, LIT80]. We then enhanced the algorithm to handle page overflows and large key handling with a single mechanism, named buddy-in-waiting.

Existing UNIX Hashing Techniques

Over the last decade, several dynamic hashing schemes have been developed for the UNIX timesharing system, starting with the inclusion of *dbm*, a minimal database library written by Ken Thompson [THOM90], in the Seventh Edition UNIX system. Since then, an extended version of the same library, *ndbm*, and a public-domain clone of the latter, *sdbm*, have been developed. Another interface-compatible library *gdbm*, was recently made available as part of the Free Software Foundation's (FSF) software distribution.

All of these implementations are based on the idea of revealing just enough bits of a hash value to locate a page in a single access. While *dbm/ndbm* and *sdbm* map the hash value directly to a disk address, *gdbm* uses the hash value to index into a *directory* [ENB88] containing disk addresses.

The *hsearch* routines in System V are designed to provide memory-resident hash tables. Since data access does not require disk access, simple hashing schemes which may require multiple probes into the table are used. A more interesting version of *hsearch* is a public domain library, *dynahash*, that implements Larson's in-memory adaptation [LAR88] of linear hashing [LIT80].

dbm and *ndbm*

The *dbm* and *ndbm* library implementations are based on the same algorithm by Ken Thompson [THOM90, TOR88, WAL84], but differ in their programmatic interfaces. The latter is a modified version of the former which adds support for multiple

databases to be open concurrently. The discussion of the algorithm that follows is applicable to both *dbm* and *ndbm*.

The basic structure of *dbm* calls for fixed-sized disk blocks (buckets) and an *access* function that maps a key to a bucket. The interface routines use the *access* function to obtain the appropriate bucket in a single disk access.

Within the *access* function, a bit-randomizing hash function¹ is used to convert a key into a 32-bit hash value. Out of these 32 bits, only as many bits as necessary are used to determine the particular bucket on which a key resides. An in-memory bitmap is used to determine how many bits are required. Each bit indicates whether its associated bucket has been split yet (a 0 indicating that the bucket has not yet split). The use of the hash function and the bitmap is best described by stepping through database creation with multiple invocations of a *store* operation.

Initially, the hash table contains a single bucket (bucket 0), the bit map contains a single bit (bit 0 corresponding to bucket 0), and 0 bits of a hash value are examined to determine where a key is placed (in bucket 0). When bucket 0 is full, its bit in the bitmap (bit 0) is set, and its contents are split between buckets 0 and 1, by considering the 0th bit (the lowest bit not previously examined) of the hash value for each key within the bucket. Given a well-designed hash function, approximately half of the keys will have hash values with the 0th bit set. All such keys and associated data are moved to bucket 1, and the rest remain in bucket 0.

After this split, the file now contains two buckets, and the bitmap contains three bits: the 0th bit is set to indicate a bucket 0 split when no bits of the hash value are considered, and two more unset bits for buckets 0 and 1. The placement of an incoming key now requires examination of the 0th bit of the hash value, and the key is placed either in bucket 0 or bucket 1. If either bucket 0 or bucket 1 fills up, it is split as before, its bit is set in the bitmap, and a new set of unset bits are added to the bitmap.

Each time we consider a new bit (bit *n*), we add 2^{n+1} bits to the bitmap and obtain 2^{n+1} more addressable buckets in the file. As a result, the bitmap contains the previous $2^{n+1}-1$ bits ($1+2+4+\dots+2^n$) which trace the entire *split history* of the addressable buckets.

Given a key and the bitmap created by this algorithm, we first examine bit 0 of the bitmap (the bit to consult when 0 bits of the hash value are being examined). If it is set (indicating that the

bucket split), we begin considering the bits of the 32-bit hash value. As bit *n* is revealed, a mask equal to $2^{n+1}-1$ will yield the current bucket address. Adding $2^{n+1}-1$ to the bucket address identifies which bit in the bitmap must be checked. We continue revealing bits of the hash value until all set bits in the bitmap are exhausted. The following algorithm, a simplification of the algorithm due to Ken Thompson [THOM90, TOR88], uses the hash value and the bitmap to calculate the bucket address as discussed above.

```
hash = calchash(key);
mask = 0;
while (isbitset((hash & mask) + mask))
    mask = (mask << 1) + 1;
bucket = hash & mask;
```

sdbm

The *sdbm* library is a public-domain clone of the *ndbm* library, developed by Ozan Yigit to provide *ndbm*'s functionality under some versions of UNIX that exclude it for licensing reasons [YIG89]. The programmer interface, and the basic structure of *sdbm* is identical to *ndbm* but internal details of the *access* function, such as the calculation of the bucket address, and the use of different hash functions make the two incompatible at the database level.

The *sdbm* library is based on a simplified implementation of Larson's 1978 *dynamic hashing* algorithm including the *refinements and variations* of section 5 [LAR78]. Larson's original algorithm calls for a forest of binary hash trees that are accessed by two hash functions. The first hash function selects a particular tree within the forest. The second hash function, which is required to be a boolean pseudo-random number generator that is seeded by the key, is used to traverse the tree until internal (split) nodes are exhausted and an external (non-split) node is reached. The bucket addresses are stored directly in the external nodes.

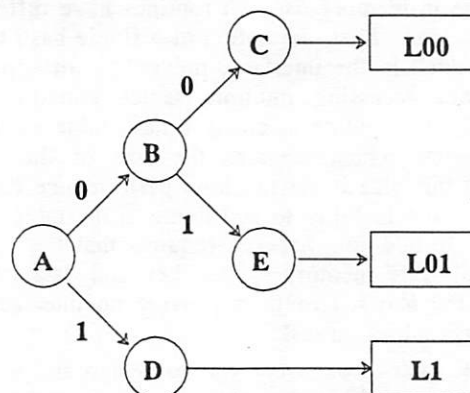


Figure 1: Radix search trie with internal nodes A and B, external nodes C, D, and E, and bucket addresses stored in the unused portion of the trie.

¹ This bit-randomizing property is important to obtain radically different hash values for nearly identical keys, which in turn avoids clustering of such keys in a single bucket.

Larson's refinements are based on the observation that the nodes can be represented by a single bit that is set for internal nodes and not set for external nodes, resulting in a radix search trie. Figure 1 illustrates this. Nodes A and B are internal (split) nodes, thus having no bucket addresses associated with them. Instead, the external nodes (C, D, and E) each need to refer to a bucket address. These bucket addresses can be stored in the trie itself where the subtrees would live if they existed [KNU68]. For example, if nodes F and G were the children of node C, the bucket address L00 could reside in the bits that will eventually be used to store nodes F and G and all their children.

Further simplifications of the above [YIG89] are possible. Using a single radix trie to avoid the first hash function, replacing the pseudo-random number generator with a well designed, bit-randomizing hash function, and using the portion of the hash value exposed during the trie traversal as a direct bucket address results in an *access* function that works very similar to Thompson's algorithm above. The following algorithm uses the hash value to traverse a linearized radix trie² starting at the 0th bit.

```

tbit = 0;      /* radix trie index */
hbit = 0;      /* hash bit index */
mask = 0;
hash = calchash(key);

for (mask = 0;
     isbitset(tbit);
     mask = (mask << 1) + 1)
    if (hash & (1 << hbit++))
        /* right son */
        tbit = 2 * tbit + 2;
    else
        /* left son */
        tbit = 2 * tbit + 1;

bucket = hash & mask;

```

gdbm

The *gdbm* (GNU data base manager) library is a UNIX database manager written by Philip A. Nelson, and made available as a part of the FSF software distribution. The *gdbm* library provides the same functionality of the *dbm/ndbm* libraries [NEL90] but attempts to avoid some of their shortcomings. The *gdbm* library allows for arbitrary-length data, and its database is a singular, non-sparse³ file. The *gdbm* library also includes *dbm* and *ndbm* compatible interfaces.

The *gdbm* library is based on *extensible hashing*, a dynamic hashing algorithm by Fagin et al [FAG79]. This algorithm differs from the previously

² A linearized radix trie is merely an array representation of the radix search trie described above. The children of the node with index *i* can be found at the nodes indexed $2*i+1$ and $2*i+2$.

³It does not contain holes.

discussed algorithms in that it uses a *directory* that is a collapsed representation [ENB88] of the radix search trie used by *sdbm*.

In this algorithm, a directory consists of a search trie of depth *n*, containing 2^n bucket addresses (i.e. each element of the trie is a bucket address). To access the hash table, a 32-bit hash value is calculated and *n* bits of the value are used to index into the directory to obtain a bucket address. It is important to note that multiple entries of this directory may contain the same bucket address as a result of directory doubling during bucket splitting. Figure 2 illustrates the relationship between a typical (skewed) search trie and its directory representation. The formation of the directory shown in the figure is as follows.

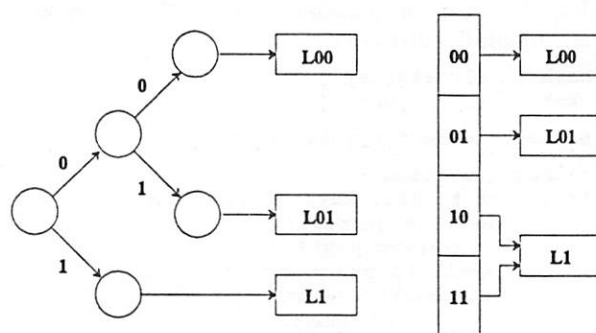


Figure 2: A radix search trie and a directory representing the trie.

Initially, there is one slot in the directory addressing a single bucket. The depth of the trie is 0 and 0 bits of each hash value are examined to determine in which bucket to place a key; all keys go in bucket 0. When this bucket is full, its contents are divided between L0 and L1 as was done in the previously discussed algorithms. After this split, the address of the second bucket must be stored in the directory. To accommodate the new address, the directory is split⁴, by doubling it, thus increasing the depth of the directory by one.

After this split, a single bit of the hash value needs to be examined to decide whether the key belongs to L0 or L1. Once one of these buckets fills (L0 for example), it is split as before, and the directory is split again to make room for the address of the third bucket. This splitting causes the addresses

⁴ This decision to split the directory is based on a comparison of the depth of the page being split and the depth of the trie. In Figure 2, the depths of both L00 and L01 are 2, whereas the depth of L1 is 1. Therefore, if L1 were to split, the directory would not need to split. In reality, a bucket is allocated for the directory at the time of file creation so although the directory splits logically, physical splits do not occur until the file becomes quite large.

of the non-splitting bucket (L1) to be duplicated. The directory now has four entries, a depth of 2, and indexes the buckets L00, L01 and L1, as shown in the Figure 2.

The crucial part of the algorithm is the observation that L1 is addressed twice in the directory. If this bucket were to split now, the directory already contains room to hold the address of the new bucket. In general, the relationship between the directory and the number of bucket addresses contained therein is used to decide when to split the directory. Each bucket has a depth, (n_b), associated with it and appears in the directory exactly 2^{n-n_b} times. When a bucket splits, its depth increases by one. The directory must split any time a bucket's depth exceeds the depth of the directory. The following code fragment helps to illustrate the extendible hashing algorithm [FAG79] for accessing individual buckets and maintaining the directory.

```
hash = calchash(key);
mask = maskvec[depth];

bucket = directory[hash & mask];

/* Key Insertion */
if (store(bucket, key, data) == FAIL) {
    newbl = getpage();
    bucket->depth++;
    newbl->depth = bucket->depth;
    if (bucket->depth > depth) {
        /* double directory */
        depth++;
        directory = double(directory);
    }
    splitbucket(bucket, newbl)
    ...
}
```

hsearch

Since *hsearch* does not have to translate hash values into disk addresses, it can use much simpler algorithms than those defined above. System V's *hsearch* constructs a fixed-size hash table (specified by the user at table creation). By default, a multiplicative hash function based on that described in Knuth, Volume 3, section 6.4 [KNU68] is used to obtain a primary bucket address. If this bucket is full, a secondary multiplicative hash value is computed to define the probe interval. The probe interval is added to the original bucket address (modulo the table size) to obtain a new bucket address. This process repeats until an empty bucket is found. If no bucket is found, an insertion fails with a "table full" condition.

The basic algorithm may be modified by a number of compile time options available to those users with AT&T source code. First, the package provides two options for hash functions. Users may specify their own hash function by compiling with "USCR" defined and declaring and defining the variable *hcompare*, a function taking two string arguments and returning an integer. Users may also

request that hash values be computed simply by taking the modulo of key (using division rather than multiplication for hash value calculation). If this technique is used, collisions are resolved by scanning sequentially from the selected bucket (linear probing). This option is available by defining the variable "DIV" at compile time.

A second option, based on an algorithm discovered by Richard P. Brent, rearranges the table at the time of insertion in order to speed up retrievals. The basic idea is to shorten long probe sequences by lengthening short probe sequences. Once the probe chain has exceeded some threshold (Brent suggests 2), we attempt to shuffle any colliding keys (keys which appeared in the probe sequence of the new key). The details of this key shuffling can be found in [KNU68] and [BRE73]. This algorithm may be obtained by defining the variable "BRENT" at compile time.

A third set of options, obtained by defining "CHAINED", use linked lists to resolve collisions. Either of the primary hash function described above may be used, but all collisions are resolved by building a linked list of entries from the primary bucket. By default, new entries will be added to a bucket at the beginning of the bucket chain. However, compile options "SORTUP" or "SORTDOWN" may be specified to order the hash chains within each bucket.

dynahash

The *dynahash* library, written by Esmond Pitt, implements Larson's linear hashing algorithm [LAR88] with an *hsearch* compatible interface. Intuitively, a hash table begins as a single bucket and grows in generations, where a generation corresponds to a doubling in the size of the hash table. The 0th generation occurs as the table grows from one bucket to two. In the next generation the table grows from two to four. During each generation, every bucket that existed at the beginning of the generation is split.

The table starts as a single bucket (numbered 0), the current split bucket is set to bucket 0, and the maximum split point is set to twice the current split point (0). When it is time for a bucket to split, the keys in the current split bucket are divided between the current split bucket and a new bucket whose bucket number is equal to 1 + current split bucket + maximum split point. We can determine which keys move to the new bucket by examining the n^{th} bit of a key's hash value where n is the generation number. After the bucket at the maximum split point has been split, the generation number is incremented, the current split point is set back to zero, and the maximum split point is set to the number of the last bucket in the file (which is equal to twice the old maximum split point plus 1).

To facilitate locating keys, we maintain two masks. The low mask is equal to the maximum split bucket and the high mask is equal to the next maximum split bucket. To locate a specific key, we compute a 32-bit hash value using a bit-randomizing algorithm such as the one described in [LAR88]. This hash value is then masked with the high mask. If the resulting number is greater than the maximum bucket in the table (current split bucket + maximum split point), the hash value is masked with the low mask. In either case, the result of the mask is the bucket number for the given key. The algorithm below illustrates this process.

```
h = calchash(key);
bucket = h & high_mask;
if ( bucket > max_bucket )
    bucket = h & low_mask;
return(bucket);
```

In order to decide when to split a bucket, *dynahash* uses *controlled splitting*. A hash table has a fill factor which is expressed in terms of the average number of keys in each bucket. Each time the table's total number of keys divided by its number of buckets exceeds this fill factor, a bucket is split.

Since the *hsearch* create interface (*hcreate*) calls for an estimate of the final size of the hash table (*nelem*), *dynahash* uses this information to initialize the table. The initial number of buckets is set to *nelem* rounded to the next higher power of two. The current split point is set to 0 and the maximum bucket and maximum split point are set to this rounded value.

The New Implementation

Our implementation is also based on Larson's linear hashing [LAR88] algorithm as well as the *dynahash* implementation. The *dbm* family of algorithms decide dynamically which bucket to split and when to split it (when it overflows) while *dynahash* splits in a predefined order (linearly) and at a predefined time (when the table fill factor is exceeded). We use a hybrid of these techniques. Splits occur in the predefined order of linear hashing, but the time at which pages are split is determined both by page overflows (*uncontrolled splitting*) and by exceeding the fill factor (*controlled splitting*).

A hash table is parameterized by both its bucket size (*bsize*) and fill factor (*ffactor*). Whereas *dynahash*'s buckets can be represented as a linked list of elements in memory, our package needs to support disk access, and must represent buckets in terms of pages. The *bsize* is the size (in bytes) of these pages. As in linear hashing, the number of buckets in the table is equal to the number of keys in the table divided by *ffactor*.⁵ The controlled

⁵ This is not strictly true. The file does not contract when keys are deleted, so the number of buckets is actually equal to the maximum number of keys ever

splitting occurs each time the number of keys in the table exceeds the fill factor multiplied by the number of buckets.

Inserting keys and splitting buckets is performed precisely as described previously for *dynahash*. However, since buckets are now comprised of pages, we must be prepared to handle cases where the size of the keys and data in a bucket exceed the bucket size.

Overflow Pages

There are two cases where a key may not fit in its designated bucket. In the first case, the total size of the key and data may exceed the bucket size. In the second, addition of a new key could cause an overflow, but the bucket in question is not yet scheduled to be split. In existing implementations, the second case never arises (since buckets are split when they overflow) and the first case is not handled at all. Although large key/data pair handling is difficult and expensive, it is essential. In a linear hashed implementation, overflow pages are required for buckets which overflow before they are split, so we can use the same mechanism for large key/data pairs that we use for overflow pages. Logically, we chain overflow pages to the buckets (also called primary pages). In a memory based representation, overflow pages do not pose any special problems because we can chain overflow pages to primary pages using memory pointers. However, mapping these overflow pages into a disk file is more of a challenge, since we need to be able to address both bucket pages, whose numbers are growing linearly, and some indeterminate number of overflow pages without reorganizing the file.

One simple solution would be to allocate a separate file for overflow pages. The disadvantage with such a technique is that it requires an extra file descriptor, an extra system call on open and close, and logically associating two independent files. For these reasons, we wanted to map both primary pages and overflow pages into the same file space.

The buddy-in-waiting algorithm provides a mechanism to support multiple pages per logical bucket while retaining the simple split sequence of linear hashing. Overflow pages are preallocated between generations of primary pages. These overflow pages are used by any bucket containing more keys than fit on the primary page and are reclaimed, if possible, when the bucket later splits. Figure 3 depicts the layout of primary pages and overflow pages within the same file. Overflow page use information is recorded in bitmaps which are themselves stored on overflow pages. The addresses of the bitmap pages and the number of pages allocated at each split point are stored in the file header.

present in the table divided by the fill factor.

Using this information, both overflow addresses and bucket addresses can be mapped to disk addresses by the following calculation:

```
int    bucket;    /* bucket address */
u_short oaddr;    /* overflow address */
int    nhdr_pages; /* npages in file header */
int    spares[32]; /* npages at each split */
int    log2();     /* ceil(log base 2) */

#define BUCKET_TO_PAGE(bucket) \
    bucket + nhdr_pages + \
    (bucket ? spares[log2(bucket + 1) - 1] : 0)

#define OADDR_TO_PAGE(oaddr) \
    BUCKET_TO_PAGE((1 << (oaddr >> 11)) - 1) + \
    oaddr & 0x7ff;
```

An overflow page is addressed by its split point, identifying the generations between which the overflow page is allocated, and its page number, identifying the particular page within the split point. In this implementation, offsets within pages are 16 bits long (limiting the maximum page size to 32K), so we select an overflow page addressing algorithm that can be expressed in 16 bits and which allows quick retrieval. The top five bits indicate the split point and the lower eleven indicate the page number within the split point. Since five bits are reserved for the split point, files may split 32 times yielding a maximum file size of 2^{32} buckets and $32 \cdot 2^{11}$ overflow pages. The maximum page size is 2^{15} , yielding a maximum file size greater than 131,000 GB (on file systems supporting files larger than 4GB).

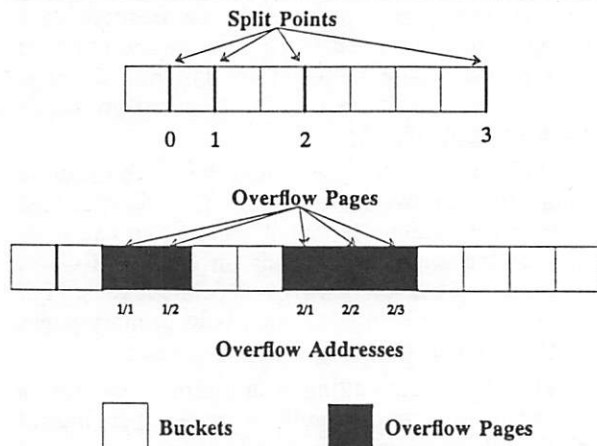


Figure 3: Split points occur between generations and are numbered from 0. In this figure there are two overflow pages allocated at split point 1 and three allocated at split point 2.

Buffer Management

The hash table is stored in memory as a logical array of bucket pointers. Physically, the array is arranged in segments of 256 pointers. Initially, there is space to allocate 256 segments. Reallocation occurs when the number of buckets exceeds 32K ($256 \cdot 256$). Primary pages may be accessed directly through the array by bucket number and

overflow pages are referenced logically by their overflow page address. For small hash tables, it is desirable to keep all pages in main memory while on larger tables, this is probably impossible. To satisfy both of these requirements, the package includes buffer management with LRU (least recently used) replacement.

By default, the package allocates up to 64K bytes of buffered pages. All pages in the buffer pool are linked in LRU order to facilitate fast replacement. Whereas efficient access to primary pages is provided by the bucket array, efficient access to overflow pages is provided by linking overflow page buffers to their predecessor page (either the primary page or another overflow page). This means that an overflow page cannot be present in the buffer pool if its primary page is not present. This does not impact performance or functionality, because an overflow page will be accessed only after its predecessor page has been accessed. Figure 4 depicts the data structures used to manage the buffer pool.

The in-memory bucket array contains pointers to buffer header structures which represent primary pages. Buffer headers contain modified bits, the page address of the buffer, a pointer to the actual buffer, and a pointer to the buffer header for an overflow page if it exists, in addition to the LRU links. If the buffer corresponding to a particular bucket is not in memory, its pointer is NULL. In effect, pages are linked in three ways. Using the buffer headers, they are linked physically through the LRU links and the overflow links. Using the pages themselves, they are linked logically through the overflow addresses on the page. Since overflow pages are accessed only after their predecessor pages, they are removed from the buffer pool when their primary is removed.

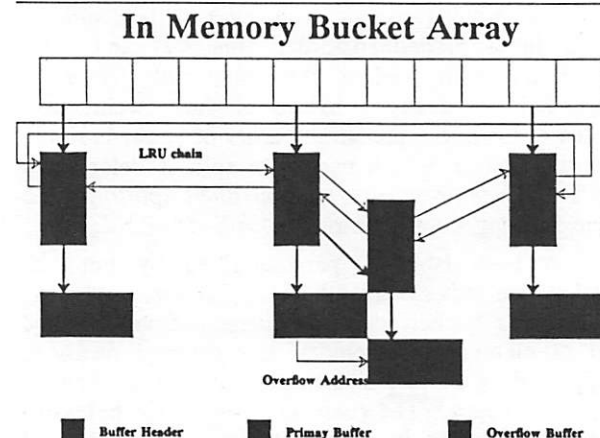


Figure 4: Three primary pages (B0, B5, B10) are accessed directly from the bucket array. The one overflow page (O1/1) is linked physically from its primary page's buffer header as well as logically from its predecessor page buffer (B5).

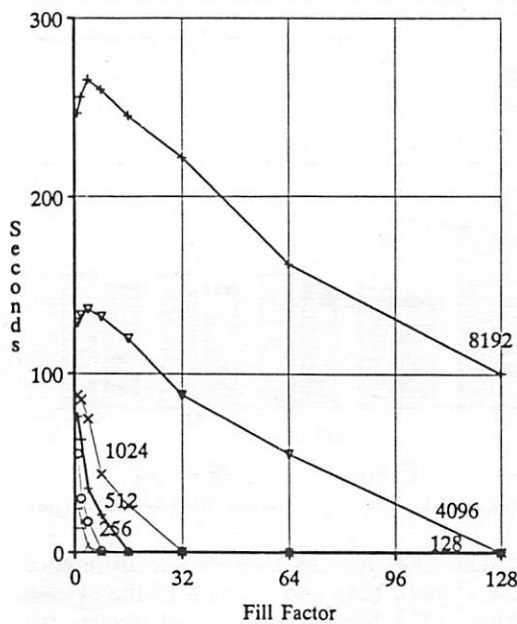


Figure 5a: System Time for dictionary data set with 1M of buffer space and varying bucket sizes and fill factors. Each line is labeled with its bucket size.

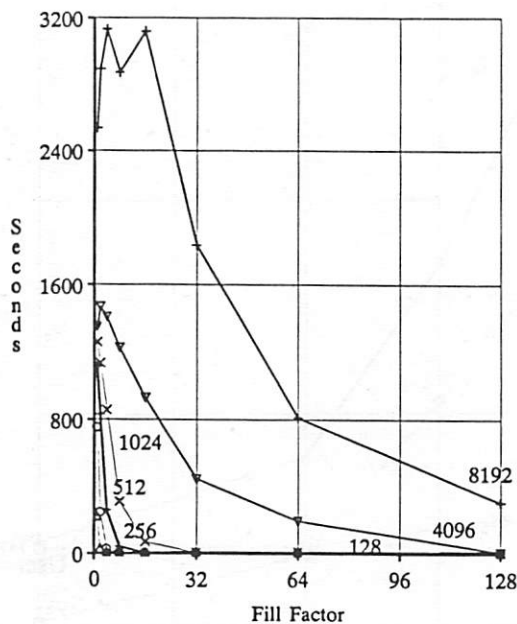


Figure 5b: Elapsed Time for dictionary data set with 1M of buffer space and varying bucket sizes and fill factors. Each line is labeled with its bucket size.

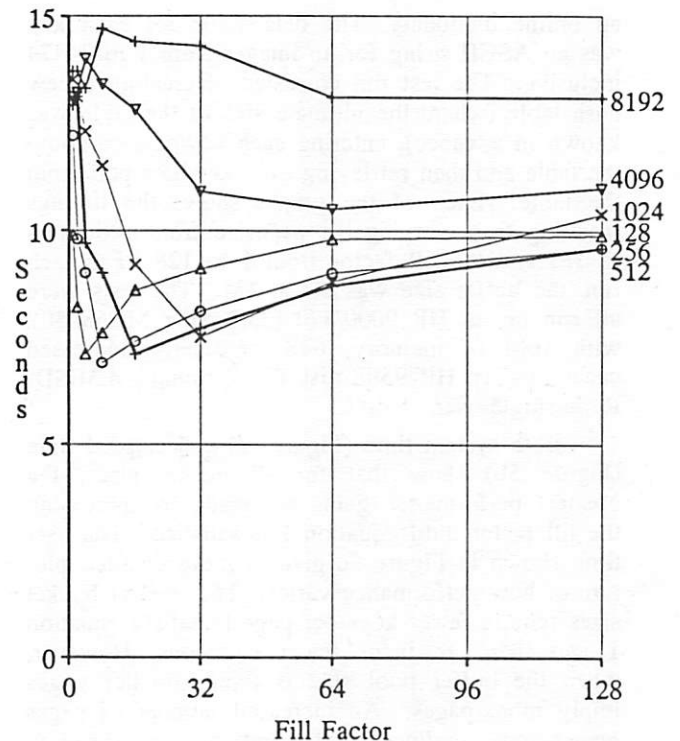


Figure 5c: User Time for dictionary data set with 1M of buffer space and varying bucket sizes and fill factors. Each line is labeled with its bucket size.

Table Parameterization

When a hash table is created, the bucket size, fill factor, initial number of elements, number of bytes of main memory used for caching, and a user-defined hash function may be specified. The bucket size (and page size for overflow pages) defaults to 256 bytes. For tables with large data items, it may be preferable to increase the page size, and, conversely, applications storing small items exclusively in memory may benefit from a smaller bucket size. A bucket size smaller than 64 bytes is not recommended.

The fill factor indicates a desired density within the hash table. It is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows. Its default is eight. If the user knows the average size of the key/data pairs being stored in the table, near optimal bucket sizes and fill factors may be selected by applying the equation:

$$(1) \quad ((\text{average_pair_length} + 4) * \text{ffactor}) \geq \text{bsize}$$

For highly time critical applications, experimenting with different bucket sizes and fill factors is encouraged.

Figures 5a,b, and c illustrate the effects of varying page sizes and fill factors for the same data set. The data set consisted of 24474 keys taken from

an online dictionary. The data value for each key was an ASCII string for an integer from 1 to 24474 inclusive. The test run consisted of creating a new hash table (where the ultimate size of the table was known in advance), entering each key/data pair into the table and then retrieving each key/data pair from the table. Each of the graphs shows the timings resulting from varying the pagesize from 128 bytes to 1M and the fill factor from 1 to 128. For each run, the buffer size was set at 1M. The tests were all run on an HP 9000/370 (33.3 Mhz MC68030), with 16M of memory, 64K physically addressed cache, and an HP7959S disk drive, running 4.3BSD-Reno single-user.

Both system time (Figure 5a) and elapsed time (Figure 5b) show that for all bucket sizes, the greatest performance gains are made by increasing the fill factor until equation 1 is satisfied. The user time shown in Figure 5c gives a more detailed picture of how performance varies. The smaller bucket sizes require fewer keys per page to satisfy equation 1 and therefore incur fewer collisions. However, when the buffer pool size is fixed, smaller pages imply more pages. An increased number of pages means more *malloc(3)* calls and more overhead in the hash package's buffer manager to manage the additional pages.

The tradeoff works out most favorably when the page size is 256 and the fill factor is 8. Similar conclusions were obtained if the test was run without knowing the final table size in advance. If the file was closed and written to disk, the conclusions were still the same. However, rereading the file from disk was slightly faster if a larger bucket size and fill factor were used (1K bucket size and 32 fill factor). This follows intuitively from the improved efficiency of performing 1K reads from the disk rather than 256 byte reads. In general, performance for disk based tables is best when the page size is approximately 1K.

If an approximation of the number of elements ultimately to be stored in the hash table is known at the time of creation, the hash package takes this number as a parameter and uses it to hash entries into the full sized table rather than growing the table from a single bucket. If this number is not known, the hash table starts with a single bucket and gracefully expands as elements are added, although a slight performance degradation may be noticed. Figure 6 illustrates the difference in performance between storing keys in a file when the ultimate size is known (the left bars in each set), compared to building the file when the ultimate size is unknown (the right bars in each set). Once the fill factor is sufficiently high for the page size (8), growing the table dynamically does little to degrade performance.

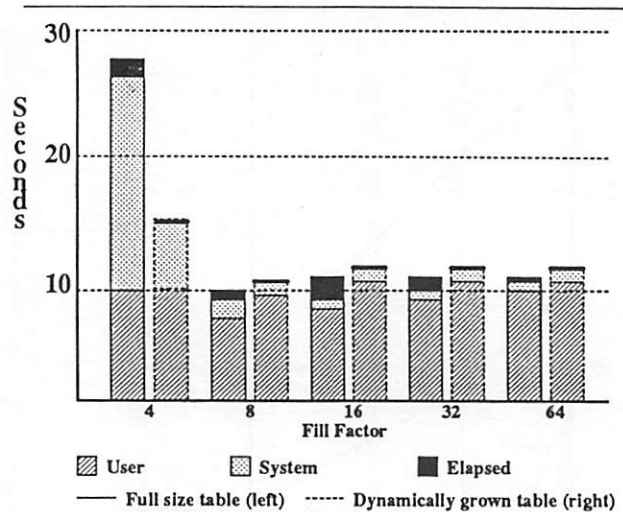


Figure 6: The total regions indicate the difference between the elapsed time and the sum of the system and user time. The left bar of each set depicts the timing of the test run when the number of entries is known in advance. The right bars depict the timing when the file is grown from a single bucket.

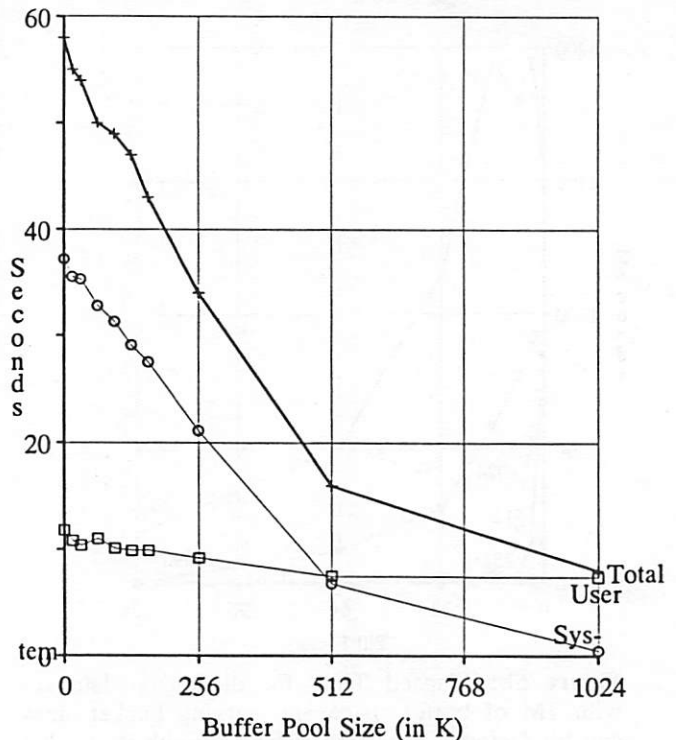


Figure 7: User time is virtually insensitive to the amount of buffer pool available, however, both system time and elapsed time are inversely proportional to the size of the buffer pool. Even for large data sets where one expects few collisions, specifying a large buffer pool dramatically improves performance.

Since no known hash function performs equally well on all possible data, the user may find that the built-in hash function does poorly on a particular data set. In this case, a hash function, taking two arguments (a pointer to a byte string and a length) and returning an unsigned long to be used as the hash value, may be specified at hash table creation time. When an existing hash table is opened and a hash function is specified, the hash package will try to determine that the hash function supplied is the one with which the table was created. There are a variety of hash functions provided with the package. The default function for the package is the one which offered the best performance in terms of cycles executed per call (it did not produce the fewest collisions although it was within a small percentage of the function that produced the fewest collisions). Again, in time critical applications, users are encouraged to experiment with a variety of hash functions to achieve optimal performance.

Since this hashing package provides buffer management, the amount of space allocated for the buffer pool may be specified by the user. Using the same data set and test procedure as used to derive the graphs in Figures 5a-c, Figure 7 shows the impact of varying the size of the buffer pool. The bucket size was set to 256 bytes and the fill factor was set to 16. The buffer pool size was varied from 0 (the minimum number of pages required to be buffered) to 1M. With 1M of buffer space, the package performed no I/O for this data set. As Figure 7 illustrates, increasing the buffer pool size can have a dramatic effect on resulting performance.⁶

Enhanced Functionality

This hashing package provides a set of compatibility routines to implement the *ndbm* interface. However, when the native interface is used, the following additional functionality is provided:

- Inserts never fail because too many keys hash to the same value.
- Inserts never fail because key and/or associated data is too large
- Hash functions may be user-specified.
- Multiple pages may be cached in main memory.

It also provides a set of compatibility routines to implement the *hsearch* interface. Again, the native interface offers enhanced functionality:

⁶ Some allocators are extremely inefficient at allocating memory. If you find that applications are running out of memory before you think they should, try varying the pagesize to get better utilization from the memory allocator.

- Files may grow beyond *nelem* elements.
- Multiple hash tables may be accessed concurrently.
- Hash tables may be stored and accessed on disk.
- Hash functions may be user-specified at runtime.

Relative Performance of the New Implementation

The performance testing of the new package is divided into two test suites. The first suite of tests requires that the tables be read from and written to disk. In these tests, the basis for comparison is the 4.3BSD-Reno version of *ndbm*. Based on the designs of *sdbm* and *gdbm*, they are expected to perform similarly to *ndbm*, and we do not show their performance numbers. The second suite contains the memory resident test which does not require that the files ever be written to disk, only that hash tables may be manipulated in main memory. In this test, we compare the performance to that of the *hsearch* routines.

For both suites, two different databases were used. The first is the dictionary database described previously. The second was constructed from a password file with approximately 300 accounts. Two records were constructed for each account. The first used the logname as the key and the remainder of the password entry for the data. The second was keyed by uid and contained the entire password entry as its data field. The tests were all run on the HP 9000 with the same configuration previously described. Each test was run five times and the timing results of the runs were averaged. The variance across the 5 runs was approximately 1% of the average yielding 95% confidence intervals of approximately 2%.

Disk Based Tests

In these tests, we use a bucket size of 1024 and a fill factor of 32.

create test

The keys are entered into the hash table, and the file is flushed to disk.

read test

A lookup is performed for each key in the hash table.

verify test

A lookup is performed for each key in the hash table, and the data returned is compared against that originally stored in the hash table.

sequential retrieve

All keys are retrieved in sequential order from the hash table. The *ndbm* interface allows sequential retrieval of the keys from the database, but does not return the data associated with each

key. Therefore, we compare the performance of the new package to two different runs of *ndbm*. In the first case, *ndbm* returns only the keys while in the second, *ndbm* returns both the keys and the data (requiring a second call to the library). There is a single run for the new library since it returns both the key and the data.

	hash	ndbm	%change
CREATE			
user	6.4	12.2	48
sys	32.5	34.7	6
elapsed	90.4	99.6	9
READ			
user	3.4	6.1	44
sys	1.2	15.3	92
elapsed	4.0	21.2	81
VERIFY			
user	3.5	6.3	44
sys	1.2	15.3	92
elapsed	4.0	21.2	81
SEQUENTIAL			
user	2.7	1.9	-42
sys	0.7	3.9	82
elapsed	3.0	5.0	40
SEQUENTIAL (with data retrieval)			
user	2.7	8.2	67
sys	0.7	4.3	84
elapsed	3.0	12.0	75

	hash	hsearch	%change
CREATE/READ			
user	6.6	17.2	62
sys	1.1	0.3	-266
elapsed	7.8	17.0	54

Figure 8a: Timing results for the dictionary database.

In-Memory Test

This test uses a bucket size of 256 and a fill factor of 8.

create/read test

In this test, a hash table is created by inserting all the key/data pairs. Then a keyed retrieval is performed for each pair, and the hash table is destroyed.

Performance Results

Figures 8a and 8b show the user time, system time, and elapsed time for each test for both the new implementation and the old implementation (*hsearch* or *ndbm*, whichever is appropriate) as well as the improvement. The improvement is expressed as a percentage of the old running time:

$$\% = 100 * (\text{old_time} - \text{new_time}) / \text{old_time}$$

	hash	ndbm	%change
CREATE			
user	0.2	0.4	50
sys	0.1	1.0	90
elapsed	0	3.2	100
READ			
user	0.1	0.1	0
sys	0.1	0.4	75
elapsed	0.0	0.0	0
VERIFY			
user	0.1	0.2	50
sys	0.1	0.3	67
elapsed	0.0	0.0	0
SEQUENTIAL			
user	0.1	0.0	-100
sys	0.1	0.1	0
elapsed	0.0	0.0	0
SEQUENTIAL (with data retrieval)			
user	0.1	0.1	0
sys	0.1	0.1	0
elapsed	0.0	0.0	0

	hash	hsearch	%change
CREATE/READ			
user	0.3	0.4	25
sys	0.0	0.0	0
elapsed	0.0	0.0	0

Figure 8b: Timing results for the password database.

In nearly all cases, the new routines perform better than the old routines (both *hsearch* and *ndbm*). Although the create tests exhibit superior user time performance, the test time is dominated by the cost of writing the actual file to disk. For the large database (the dictionary), this completely overwhelmed the system time. However, for the small database, we see that differences in both user and system time contribute to the superior performance of the new package.

The read, verify, and sequential results are deceptive for the small database since the entire test ran in under a second. However, on the larger database the read and verify tests benefit from the caching of buckets in the new package to improve performance by over 80%. Since the first sequential test does not require *ndbm* to return the data values, the user time is lower than for the new package. However when we require both packages to return data, the new package excels in all three timings.

The small database runs so quickly in the memory-resident case that the results are uninteresting. However, for the larger database the new package pays a small penalty in system time because it limits its main memory utilization and swaps pages

out to temporary storage in the file system while the *hsearch* package requires that the application allocate enough space for all key/data pair. However, even with the system time penalty, the resulting elapsed time improves by over 50%.

Conclusion

This paper has presented the design, implementation and performance of a new hashing package for UNIX. The new package provides a superset of the functionality of existing hashing packages and incorporates additional features such as large key handling, user defined hash functions, multiple hash tables, variable sized pages, and linear hashing. In nearly all cases, the new package provides improved performance on the order of 50-80% for the workloads shown. Applications such as the loader, compiler, and mail, which currently implement their own hashing routines, should be modified to use the generic routines.

This hashing package is one access method which is part of a generic database access package being developed at the University of California, Berkeley. It will include a btree access method as well as fixed and variable length record access methods in addition to the hashed support presented here. All of the access methods are based on a key/data pair interface and appear identical to the application layer, allowing application implementations to be largely independent of the database type. The package is expected to be an integral part of the 4.4BSD system, with various standard applications such as *more(1)*, *sort(1)* and *vi(1)* based on it. While the current design does not support multi-user access or transactions, they could be incorporated relatively easily.

References

- [ATT79] AT&T, DBM(3X), *Unix Programmer's Manual, Seventh Edition, Volume 1*, January, 1979.
- [ATT85] AT&T, HSEARCH(BA_LIB), *Unix System User's Manual, System V.3*, pp. 506-508, 1985.
- [BRE73] Brent, Richard P., "Reducing the Retrieval Time of Scatter Storage Techniques", *Communications of the ACM*, Volume 16, No. 2, pp. 105-109, February, 1973.
- [BSD86] NDBM(3), *4.3BSD Unix Programmer's Manual Reference Guide*, University of California, Berkeley, 1986.
- [ENB88] Enbody, R. J., Du, H. C., "Dynamic Hashing Schemes", *ACM Computing Surveys*, Vol. 20, No. 2, pp. 85-113, June 1988.
- [FAG79] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, H. Raymond Strong, "Extendible Hashing -- A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems*, Volume 4, No. 3., September 1979, pp

315-34

- [KNU68] Knuth, D.E., *The Art of Computer Programming Vol. 3: Sorting and Searching*, sections 6.3-6.4, pp 481-550.
- [LAR78] Larson, Per-Ake, "Dynamic Hashing", *BIT*, Vol. 18, 1978, pp. 184-201.
- [LAR88] Larson, Per-Ake, "Dynamic Hash Tables", *Communications of the ACM*, Volume 31, No. 4., April 1988, pp 446-457.
- [LIT80] Witold, Litwin, "Linear Hashing: A New Tool for File and Table Addressing", *Proceedings of the 6th International Conference on Very Large Databases*, 1980.
- [NEL90] Nelson, Philip A., *Gdbm 1.4 source distribution and README*, August 1990.
- [THOM90] Ken Thompson, private communication, Nov. 1990.
- [TOR87] Torek, C., "Re: dbm.a and ndbm.a archives", *USENET newsgroup comp.unix* 1987.
- [TOR88] Torek, C., "Re: questions regarding databases created with dbm and ndbm routines" *USENET newsgroup comp.unix.questions*, June 1988.
- [WAL84] Wales, R., "Discussion of "dbm" data base system", *USENET newsgroup unix.wizards*, January, 1984.
- [YIG89] Ozan S. Yigit, "How to Roll Your Own Dbm/Ndbm", *unpublished manuscript*, Toronto, July, 1989

Margo I. Seltzer is a Ph.D. student in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. Ms. Seltzer received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983. In her spare time, Margo can usually be found preparing massive quantities of food for hungry hoards, studying Japanese, or playing soccer with an exciting Bay Area Women's Soccer team, the Berkeley Bruisers.



Ozan (Oz) Yigit is currently a software engineer with the Communications Research and Development group, Computing Services, York University. His formative years were also spent at York, where he held system programmer and administrator positions for various mixtures of UNIX systems starting with Berkeley 4.1 in 1982, while at the same time obtaining a degree in Computer Science. In his copious free time, Oz enjoys working on whatever software looks interesting, which often includes language interpreters, preprocessors, and lately, program generators and expert systems. Oz has authored several public-domain software tools, including an nroff-like text formatter *proff* that is apparently still used in some basement PCs. His latest obsessions include the incredible programming language Scheme, and Chinese Brush painting.



Evolutionary Path to Network Storage Management

Robert K. Israel, Antony W. Foster, Arun Taylor,
Tracy M. Taylor, Neil Webber – Epoch Systems, Inc.

ABSTRACT

A major problem facing network administrators and users today is the management of network storage. Storage demands continue to increase as more powerful workstations become available and more sophisticated applications are developed for those workstations. Storage management tools have not kept pace with the proliferation of disk storage on workstation networks or with the increased power and number of workstations on the typical network. Network administrators and workstation users are now faced with the problem of backing up workstation disks, archiving old data when these disks fill up, and locating data distributed around a network.

In this paper we describe a storage management architecture, called the InfiniteStorage Architecture, that defines an evolutionary approach to automating the storage management of networked UNIX environments. This architecture provides for the management of several types of storage, e.g., magnetic and optical disks, tape, etc., in a storage hierarchy such that data anywhere on the network can be backed up and automatically moved to optimize the tradeoff between cost and accessibility.

The implementation of a primary element of this architecture, the Renaissance InfiniteStorage Manager, is described in detail. Based on the IEEE Mass Storage Reference Model, the InfiniteStorage Manager uses a mass storage server as a backing store for magnetic disks attached to workstations and workgroup servers on a network. The contents of the least recently used file on the network are automatically migrated to the storage server while the transparency of access is preserved.

Introduction

As the use of networked workstations has grown, so has the amount of data supporting and resulting from their applications. Increasingly data-intensive applications include electronic publishing, CASE, CAD/CAM, image processing, and data collection and analysis. Although the capacity of storage media such as Winchester disks and erasable optical diskettes has grown along with the data demand, raw capacity is not sufficient to handle the ever increasing storage problem. New facilities for managing the flood of data are needed, including automatic migration of data to trade accessibility against storage cost, automatic online backup and recovery, resource control, accounting, and security management. Benefits will be seen in the lowering of storage hardware costs, administrative costs, and the improvement of user productivity.

This paper describes a set of products developed by Epoch Systems to address the storage management problem for networks of workstations and file servers. The first product, the *InfiniteStorage Server*, is an NFS file server that provides vast amounts of low cost storage by transparently integrating magnetic disks and optical disk jukeboxes. Fully automated data migration, backup, and media management significantly reduce the

storage management effort for users and administrators. Two new products, *Renaissance InfiniteStorage* and *Renaissance Backup*, extend these automated data migration and backup facilities to the entire network.

The Motivation for Storage Management

In the past most storage problems have been limited to dealing with a chronic lack of storage space and keeping a regular backup schedule. Additional problems have arisen due to the mass volume of storage now available and the proliferation of storage devices with varying performance, access methods, and cost. This section enumerates some of the problems that network storage management systems are being designed to solve.

Lack of Online Storage

Current workstation applications use files on magnetic disk as the sole means of long-term storage. Since magnetic disks have finite capacity and are not extensible, problems develop as the disks fill up.

- Applications fail, sometimes without notice, when filesystems become full.
- Selection of files to move to an archival (offline)

medium such as magnetic tape can be difficult and time consuming.

- Location and retrieval of files from archival storage can be so difficult that it is avoided unless absolutely necessary.
- The administrative effort required to support a network of systems tends to grow in proportion to the number of disks on the network.

Accessibility of Centralized Storage

One approach to limiting the administrative overhead and aggregate cost of network storage is to centralize most storage on a small number of shared file servers. Although this has been successful in many environments, there are several problems with this approach.

- Administrative control over storage must be surrendered to the central authority. This includes both control over resource allocation and access security.
- Centralized file servers represent single points of failure and are often performance bottlenecks. Hence, during periods of server/network outage or high load, the entire organization can be paralyzed.
- File servers are often shared by more than one administrative group, resulting in increased overhead in controlling and accounting for resource use.
- The capability for growth of traditional file servers is limited by considerations such as the time to backup filesystems, the time to check and repair filesystems during a system boot, and the bandwidth of the network interface.

Network-wide Backups

Current backup tools have been developed either for standalone systems or for small networks. As the size of networks and storage capacity grows, backing up the entire network becomes a formidable effort.

- Tape drives or other devices traditionally used for backups are usually scarce resources, creating a major problem for scheduling and coordinating the backup of disks on the network.
- The traditional backup tools for workstations and file servers demand human intervention to take filesystems offline and handle tape volumes, and require a separate set of tape volumes for each system.
- The resources (time and media) needed to perform backups grow in proportion to the number of disks on the network, i.e., there is no economy of scale.
- The regimentation involved in manually running a regular program of backups, coupled with the

complacency that often sets in after a few months of failure-free operation, often results in backups not being run in a timely manner.

It is not surprising, given this environment, that in many cases backups of individual workstations are not done at all. Many organizations use workstation disks only for paging space, temporary files, and standard system files ("dataless" workstations), so that these disks do not have to be backed up. Others require users to develop *ad hoc* schemes such as manually creating redundant copies of important files on a regular basis.

Related Work

The storage management problems described above have been developing for some time, and Epoch Systems has not been alone in attempting to address them. This section describes relevant work in the areas of distributed filesystems, data migration, and mass storage systems.

Network File System

The Network File System [Sun86] (NFS) developed by Sun Microsystems has become a *de facto* file sharing standard for workstation networks. Like most distributed filesystems, the main goals of NFS are to increase the storage available to individual systems, reduce the aggregate storage required for a network, and improve the coordination of network users through the sharing of data.

From a storage management standpoint, NFS has enabled facilities to plan and manage storage on a network-wide basis. Magnetic disks can be placed anywhere on the network and still be made available to any client system. Data that are common to many systems can be put in a single place, eliminating wasteful redundancy. Usually the majority of storage is provided by a small group of specialized file servers so that storage management is focused in a single place. Many of the new workstation applications are now workable primarily because NFS is available to help deal with the data they produce.

Although NFS enables the centralization of storage and its administration, several properties of NFS actually encourage the dispersion of storage throughout the network. NFS is a fairly network-intensive protocol; despite aggressive caching of information on client systems, a small number of workstations can easily saturate an Ethernet network with NFS traffic. The primary solutions for limiting network load are the use of local disks on client workstations and the partitioning of networks into workgroup segments with separate file servers. Both approaches complicate storage management. Other factors such as the desire of users to control their own resources and the superior performance of local disk access also tend to increase the distribution of storage. NFS makes it easier to justify buying

additional workstations and disks since the new resources can still be shared on the network.

NFS provides no storage management tools other than the sharing mechanism. Hence, facilities must develop their own schemes for backing up filesystems and managing disks that are near capacity.

Andrew

The Andrew file system (AFS) [Morris86] was developed at Carnegie Mellon University as part of a far-reaching project for supporting the educational computing needs of the university. In the AFS model all shared data are kept on a few large, centrally administered file servers. Client systems use local magnetic disks both for private data and to cache shared data from the central file servers. AFS client systems do not function as servers.

From a storage management standpoint AFS differs significantly from NFS in several areas. In AFS all sharable data is kept on departmental or enterprise-wide fileserver systems. This relieves the client users of the burden of managing their local disk storage (unless private data is kept at the client systems) at the cost of introducing scale problems on the servers. Because the caching mechanism used by AFS significantly reduces the network traffic required for file sharing, the performance limitations of centralizing storage are greatly reduced.

The AFS *volume* mechanism also assists in storage management. Volumes provide a mechanism for partitioning data into separately manageable subsets. AFS volumes are conceptually similar to NFS filesystems in that they provide storage for files, implement a hierarchical namespace, and can be mounted into an existing namespace. In contrast, however, volumes are not tied to a specific server or disk partition, but can be freely moved between disks and servers as storage needs dictate. Storage consumption by individual volumes can be controlled, expanded as needed, and used as the basis for accounting. Volumes can be replicated (*cloned*) while in active use, creating a stable copy for backup to tape. Typically, the most recent clone of a volume is kept as a "hot standby" to substitute for the original volume in the event of server or disk failure.

Although AFS provides better scaling of performance for distributed file access [Howard88], it does not completely eliminate the use of private local data on client systems. Local data provides the advantages of better performance with local control over security and resource allocation. AFS provides no support for managing these systems.

AFS also introduces considerable management problems because of the mass centralization of storage. Total shared storage on the network is limited by the magnetic disk space available on the

servers. Hence, as storage needs grow the magnetic storage on the servers must be expanded. Attempts to avoid the single point of failure problem by replicating volumes adds to the storage load. The cost of performing backups and recovering from system failures grows in proportion to the size of storage, and eventually runs into hard limits such as the number of hours in a day.

BUMP

The BRL/USNA Migration Project (BUMP) [Muuss88] was developed jointly by the US Army Ballistic Research Laboratory and the US Naval Academy to provide mechanisms for managing finite magnetic disk resources on a UNIX system via automatic archiving and retrieval of data. BUMP prevents a disk filesystem from filling up by migrating files to a secondary storage medium such as magnetic tape. The directory entry and inode for a migrated file are kept in the filesystem so that directory lookups and *stat()* calls will still work on the file. When a process attempts to open a migrated file, it is put to sleep while the file is migrated back in from secondary storage. Also, when a process attempts to write to a full filesystem, it is again put to sleep while the migration system is notified. The process is allowed to proceed once free space has been made available. Hence, except for the access delay, the migration process is transparent to filesystem users.

The key to BUMP's operation is a set of modifications to the standard filesystem source code that detect events critical to file migration. These events are communicated to a user level daemon process through a pseudo-device driver interface. The particular events communicated include the read, write, truncation, deletion, and execution of a migrated file, and filesystem low space and out of space conditions. The pseudo-device driver also provides some low level inode and file block map manipulation primitives required by the migration service. The bulk of the activity, however, is carried out by the user level daemon and utilities. These utilities scan the filesystem selecting files to migrate, handle the mechanics of copying files to and from secondary storage volumes, and maintain a database of the migrated file locations.

BUMP goes a long way towards relieving storage management problems on a standalone computer system. It provides all of the benefits of traditional archiving systems, while relieving the administrator of the tasks of selecting files to archive and keeping track of which files have been archived where. For the user, the retrieval of a file from secondary storage involves nothing more than opening and reading it. The backup process for the complete system is also improved relative to what it would have been if all data were kept on magnetic disks. Because the migration utilities automatically

make two copies of each migrated image, filesystem backups only need to save disk-resident data.

IEEE Mass Storage Reference Model

The IEEE Mass Storage System Reference Model [IEEE90] is currently under development by the IEEE Technical Committee on Mass Storage Systems and Technology. The model provides a framework for describing the functionality required of mass storage systems, not a specific implementation architecture. By providing a consistent set of concepts and terminology, the reference model lays the foundation for the development of standard mass storage architectures and interfaces.

The reference model partitions a mass storage environment into the following logical entities:

- The *Bitfile Client* presents an application-oriented storage abstraction, defining concepts such as *files*, *directories*, *file attributes*, and *access control*.
- The *Bitfile Server* provides the storage needed to implement the Bitfile Client's abstract view. The Bitfile Server manages objects called *bitfiles*, which contain uninterpreted data and attributes and are identified by globally unique identifiers called *bitfile IDs*.
- The *Name Server* provides a mapping between application-oriented file names and the IDs of the bitfiles used to hold the files' data.
- The *Storage Server* provides a set of perfect (defect-free) *logical volumes*, which are the storage containers used by the Bitfile Server to hold bitfiles. These logical volumes may have associated properties such as size and location.
- The *Bitfile Mover* is a bulk data movement service, which may be used by the Bitfile Client or the Bitfile Server to transfer large quantities of data among logical volumes and applications.
- The *Physical Volume Repository* manages the real physical media used to implement logical volumes. Its tasks include physical volume identification, access control, jukebox control, and physical device access.
- The *Site Manager* provides tools for monitoring and controlling the actions of the other services.

Pending the development of a standard architecture based on the reference model, the model's primary use has been in describing and comparing existing mass storage systems [Arneson88]. As an example consider the UNIX NFS architecture. The Bitfile Client corresponds to the client NFS virtual file system, which implements the application interface to the service (i.e., the vnode operations). The Bitfile Server is provided by the NFS server daemons through the NFS protocol primitives *read*, *write*, *create*, *remove*, *getattr*, and *setattr*. The Name Server is also provided by the NFS server

daemons, through the protocol primitives *readdir*, *lookup*, *create*, *rename*, *remove*, *link*, *symlink*, *readlink*, *mkdir*, and *rmdir*. The Bitfile Mover function is embedded in the protocol primitives *read* and *write*. The Storage Server and Physical Volume Repository are services embedded in the host operating system of the NFS server. The Site Manager functions include the server's filesystem export utility, the client mount utility, and monitoring tools such as *nfsstat* and *rpcinfo*.

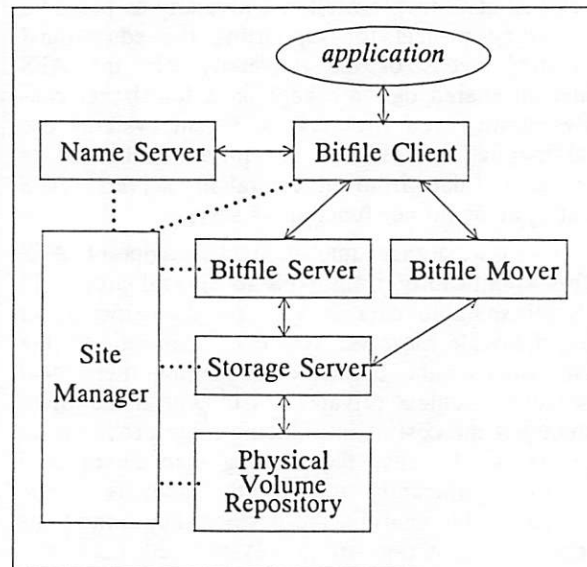


Figure 1: IEEE Reference Model

The InfiniteStorage Architecture

Epoch Systems is developing a family of tools for managing storage for networks of workstations and file servers. [Kenley90] The initial product was the InfiniteStorage Server, an NFS file server that combines magnetic disks and optical disk jukeboxes to provide managed storage up to 1 terabyte. This product has been described in detail elsewhere [Taylor89], a summary of the ideas behind the server is presented below.

A new product, the Renaissance InfiniteStorage Manager, extends the data migration concepts developed for the InfiniteStorage Server to an entire network of UNIX workstations. The implementation and use of this tool are described in detail. A companion product, Renaissance Backup, provides high performance and high capacity backup and recovery services for workstation networks. This product is currently under development and is described briefly.

Epoch-1 InfiniteStorage Server

The Epoch-1 InfiniteStorage Server is a high capacity, high performance NFS file server. Its purpose is to allow the centralized concentration of shared data on the network without incurring the

storage management overhead and cost of a large number of traditional file servers. Aside from significant efforts invested in providing high performance network [Israel89] and magnetic filesystem services, the two primary storage management facilities provided by the server are hierarchical data migration and automated online backup and recovery services.

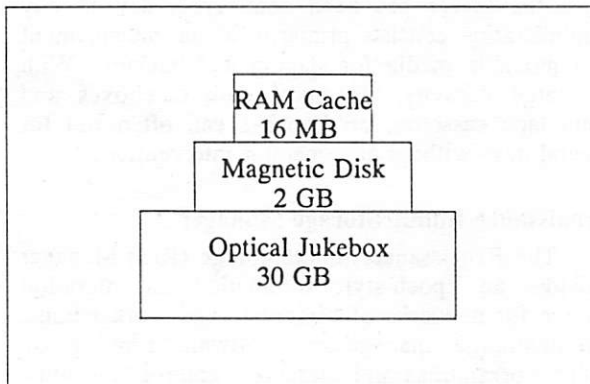


Figure 2: Storage Hierarchy

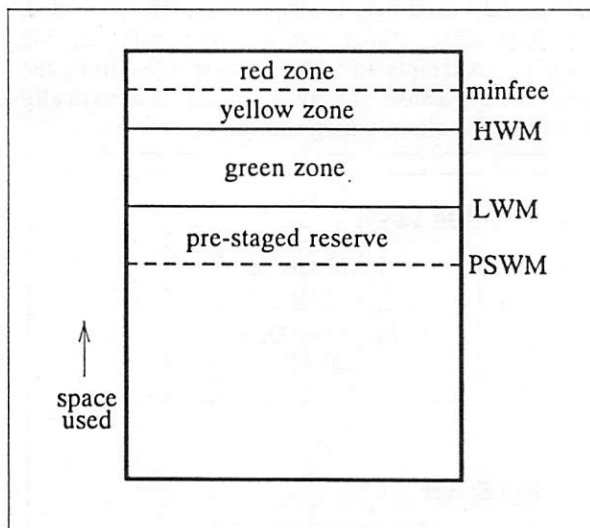


Figure 3: ISM Water Marks

Storage on the Epoch-1 is provided by a combination of magnetic disks, optical disk jukeboxes, and 8mm helical-scan tape cassettes. The InfiniteStorage Manager (ISM) integrates all of these resources transparently by moving data from one medium to another to optimize the tradeoff between access time and storage cost. Active files are normally kept in an enhanced BSD-style filesystem on magnetic storage. The goal of ISM is to keep this magnetic storage utilization between configurable *high* and *low water marks* (see Figure 3). When magnetic filesystem utilization reaches the high water mark, ISM automatically *stages* (moves) the least recently used (LRU) data from the magnetic disk to the next level of storage, usually an erasable optical disk, until

utilization drops to the low water mark (Figure 4). Even after utilization reaches the low water mark ISM continues to stage file data to the next storage level, but without deleting the magnetic disk copy. These files provide the *prestaged reserve*, a pool of magnetic space that can be freed quickly without requiring a stage-out. The *prestage water mark* defines the size of the prestaged reserve.

In addition to the *demand staging* that occurs when utilization crosses the high water mark, ISM also performs *periodic staging*. At configured times (usually early morning) ISM drives all magnetic filesystem utilization down to the low water mark. This provides a cushion of available magnetic storage for the next day's work, so that in normal operation demand staging never occurs.

The naming and attribute information for staged files remains in the magnetic filesystem, so that NFS operations such as *lookup* and *getattr* do not reference the secondary levels of storage. A sophisticated volume management system, integrated with device and jukebox management, permits the allocation and reference of secondary storage media with minimal administrator or operator intervention. In most cases, the combination of LRU staging policy and the rapid mounting of volumes by jukeboxes makes the entire spectrum of server storage appear to be online.

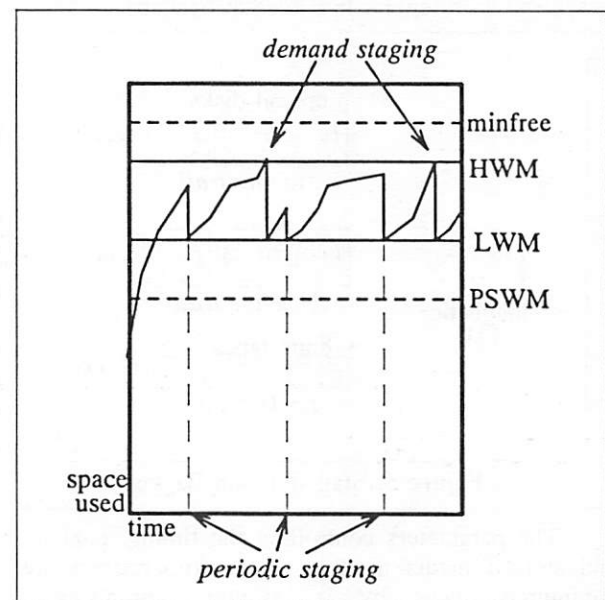


Figure 4: Staging Over Time

If a staged file is read or written, ISM moves the file back onto magnetic storage before performing the operation. For read operations the reference to the staged data is preserved, allowing ISM to reclaim the magnetic space for the file later without any access to secondary storage. For write operations the reference to the staged data is deleted, so the changed data is resident solely on magnetic disk.

ISM does not support the alteration of existing staged data because modification of this data is not possible on most forms of secondary mass storage (e.g., tape or WORM opticals). As staged files are modified or deleted, significant portions of the secondary storage media become unreferenced (*stale*). To avoid the resulting loss of storage capacity, utilities are provided to regularly compact and eliminate unreferenced data in the secondary storage media.

The Epoch-1 also includes a sophisticated backup and recovery system that works in conjunction with ISM to guarantee the integrity of all the data on the server without service interruptions or high administrative overhead. Backing up the server's data involves two operations:

- *Baseline staging.* Files that have been staged since the last backup run are also staged to a second set of storage media, called a *baseline trail*. These media are intended to be transported to off-site storage.
- *Magnetic backup.* The contents of the magnetic filesystems are collected into a serial data stream called a *saveset*, which is written to a separate set of mass storage media called a *backup trail*. A multi-level backup model is used (similar to that of BSD *dump/restore*) that allows for occasional complete backups of the magnetic data interspersed with regular incremental backups.

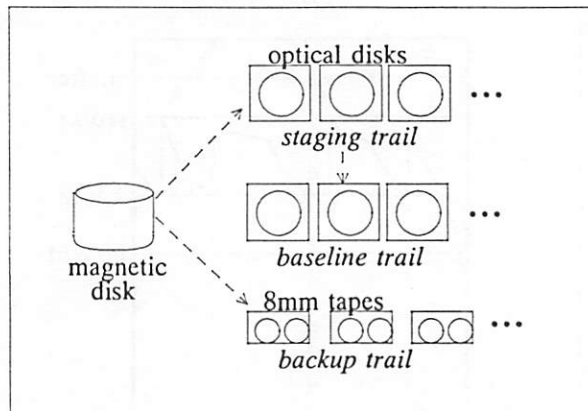


Figure 5: Stage-out and Backup

The parameters controlling the timing, content, and storage media used in the backup process are configured once by a system administrator. Thereafter backups run automatically and prompt the operator when volumes need to be mounted or formatted. The backup system operates through the standard filesystem interface while the filesystem is online, so that user access is not interrupted. Changes made to the filesystem while a backup is running are detected and recorded in the backup *saveset*. Information about each backed up file and filesystem is automatically cataloged in a database, eliminating the need for the administrator to keep

track of individual backup media. The recover facility uses this catalog to automatically identify the location of files in the backup media. Both the restoration of individual files and the reconstruction of a directory hierarchy as it existed at a given point in time are supported.

The Epoch-1 significantly reduces the effort required to manage centralized mass data storage. Once the server has been configured, day to day administration consists primarily of the management of removable media for staging and backup. With the large capacity of optical disk jukeboxes and 8mm tape cassettes, an Epoch-1 can often run for several days without any operator intervention.

Renaissance InfiniteStorage Manager

The Renaissance InfiniteStorage (RIS) Manager provides an Epoch-style automatic data migration service for networks of file servers and workstations. InfiniteStorage management software running on client workstations and file servers controls the utilization of magnetic disk space on those systems. When a filesystem reaches its high water mark, the least recently used data is staged from the client disk to a RIS mass storage server (currently an the Epoch-1). Attempts to access staged files from the client system cause the data to be automatically restored to the client's magnetic disk.

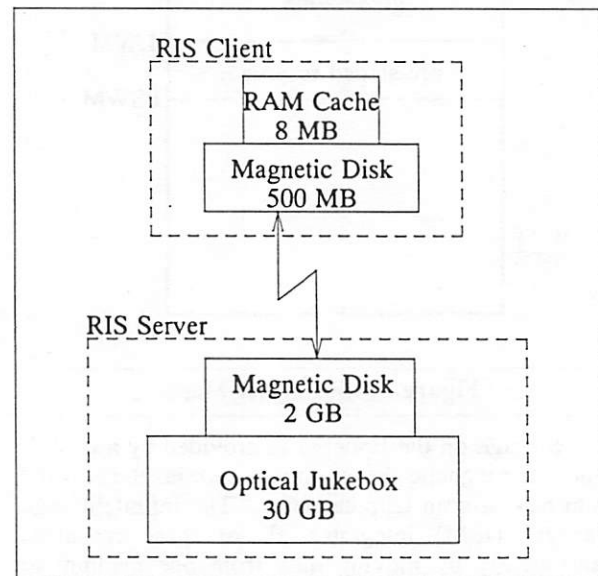


Figure 6: RIS Storage Hierarchy

The RIS server provides a central pool of secure, managed storage for RIS client systems. The unit of storage is called a *bitfile*, which is an uninterpreted byte vector of arbitrary length. Bitfiles are created within logical repositories called *client stores*. A client store is devoted to a specific RIS client system, which has sole responsibility for the allocation and use of bitfiles within the store.

Primitives are provided, via the RIS Protocol, for the client to create, delete, and read bitfiles. In addition, support is provided to coordinate backup and recovery and to synchronize bitfile reference counts between the client and server.

The initial RIS client implementation is designed to provide InfiniteStorage services for UNIX magnetic disk filesystems accessed through a generic filesystem interface such as Sun's VFS [Kleiman86]. No modifications to existing UNIX kernel software are required. The low level "hooks" for ISM are provided by interposing a thin *wrapper* layer between the VFS layer and the native filesystem implementation. Rather than modify the inode structure of the filesystem, the extended attributes needed for ISM are kept in an auxiliary file on each filesystem. ISM keeps track of the files that require special processing so that accesses to magnetic-resident files immediately fall through to the native filesystem code.

Renaissance Backup

The Renaissance Backup (RB) product currently being developed provides automated, online backup and recovery services for networks of file servers and workstations. RB does not depend on RIS services to function. In the initial release RB consists of a centralized backup and recovery manager running on an Epoch-1 file server. Client filesystems are backed up using the NFS protocol, with backup savesets and catalogs being managed in the same way as the standard Epoch-1 backup. For RIS clients, additional primitives needed for backup are provided by a secondary protocol tentatively called the XNFS protocol. The primary functions provided include the ability to retrieve the extended staging attributes for a file, and to read a file without affecting the access time (which is used in the LRU calculations by ISM).

Although an NFS-based backup is adequate for most networks, the performance limitations of file hierarchy traversals over the network limit the scalability of this approach. Epoch is currently investigating methods of performing the tree traversal and backup saveset creation on the client systems, allowing the RB server to focus on high volume processing of savesets and catalogs.

Renaissance InfiniteStorage Implementation

RIS was designed to fit easily and transparently into existing UNIX workstation environments. Since the RIS client must be portable to many environments, its implementation avoids modification of vendor-supplied software by using widely supported interfaces such as UNIX character devices and VFS. To maintain flexibility and compatibility, the on-disk filesystem structures were not altered. All storage management functions were required to operate

automatically without interfering with the user's current usage patterns or requiring special treatment from applications. Security and integrity of a user's data could not be compromised.

The following subsections describe the design of the initial RIS implementation developed for Sun client systems with the Epoch-1 as the server. At this writing, client and server ports to other vendor platforms are in progress.

RIS Protocol

Many of the fundamental concepts of the RIS architecture are expressed in the RIS Protocol (RISP), an RPC protocol that provides the primary interface between the RIS client and server systems. The initial RISP implementation uses SunRPC [RPC86] over TCP/IP. Some customization was done to the RPC interface library (but not the over-the-wire protocol) to avoid data copies in XDR [XDR86] processing and set the TCP transmission and buffering parameters to optimize bulk data transfer. SunRPC was chosen primarily because of its widespread availability. TCP was chosen because recent improvements in TCP transmission algorithms have provided acceptable performance and because SunRPC over UDP is difficult to use for non-idempotent protocol requests.

The main conceptual objects manipulated in RISP are *client stores* and *bitfiles*. A client store (hereafter referred to simply as a "store") is named using a universally unique identifier that is assigned at creation and cannot be changed. A store is located on a single server at any point in time, but it can be moved from one server to another. A store "belongs" to a single RIS client system, which is solely responsible for managing the bitfiles within that store. Other client systems may be permitted read-only access to the store, but the ability to create or delete bitfiles is restricted to the owning client. Usage quotas can be applied to stores to limit both the total storage devoted to the store and the number of bitfiles within the store. The client can determine the current usage and quota values using a `store_status` request.

Bitfiles are also named using unique identifiers, but in this case the bitfile identifier is unique only within the context of the store in which it is created. Thus, a bitfile is fully identified by the concatenation of store identifier and bitfile identifier, otherwise known as the *staging ID*. The bitfile ID is assigned by the server during creation and cannot be altered thereafter. One or more bitfiles are created using the `create_bitfiles` request. After the initial creation, the bitfile is considered to be *incomplete* until all of its data has been provided by the client using `write_bitfiles` requests. When the final write is performed, the bitfile becomes *immutable* and is committed to nonvolatile storage. Only at that time can

the client discard its copy of the data stored in the bitfile. For stage-in operations, the client obtains data from a bitfile using the `read_bitfiles` request. Any portion of any bitfile may be read at any time, although the staging application is heavily optimized for sequential access. The create, write, and read operations all allow multiple bitfiles to be accessed in a single request, and data can be requested/provided in arbitrary sizes (limited to 64KB per request in the current implementation).

The server maintains reference counts for bitfiles. The reference count of a bitfile is decremented using the `delete_bitfiles` request and incremented using the `undelete_bitfiles` request. Normally if the reference count on a bitfile drops to zero the bitfile is removed from the store on the server. If an undelete is requested for a missing bitfile, the bitfile is automatically scheduled for recovery from the server's backup system (refer to the Client Backup section).

Despite the reference count maintenance mechanism, occasional server or network outages during normal client operation can result in unreferenced bitfiles accumulating in the stores. To assist in identifying and eliminating these files, the protocol provides the `enumerate_bitfiles` primitive to allow the client to obtain a complete list of a store's bitfiles and their reference counts. The client can compare this list against its local set of bitfile references and adjust erroneous server counts using delete and undelete requests.

All RISP requests allow multiple bitfiles to be specified in each operation. There are no protocol-defined limits on the number of bitfiles per request or the amount of data per request. In practice, particular client and server implementations will have inherent limitations on the sizes of requests that can be handled. RISP avoids compatibility problems by allowing the server to advertise its limits to the client via a `server_status` request. The client controls the size of the replies it receives via the values provided in the individual requests. Thus, clients and servers are automatically able to adjust to each other's limits.

RIS Server

The RIS server has two primary functions: servicing RISP requests and managing the huge volume of storage placed on it. In the Epoch-1 RIS implementation bitfiles and stores are implemented within the magnetic filesystem framework, allowing the server's ISM and backup systems to take over management of the data. RIS-specific storage management is thus reduced to the creation and maintenance of client stores.

A client store is implemented as a directory within the server file tree. When a store is created the administrator provides a pathname indicating the

location of the store, a character string name to be associated with the store, and the owning client for the store. The pathname for the store's directory is created by appending the store name to the path provided. Within that directory, the following files and directories are created:

- *store_conf_db* - a file containing store-specific configuration information. This file defines the store identifier, the owning client, a local identifier used for quota management, etc.
- *store_state_db* - a file containing state information that must be shared between multiple active server processes. Its primary purpose is to hold an index counter used in bitfile ID generation.
- *recover_list* - a file containing a list of bitfile IDs that must be retrieved from the system backup, as a result of an undelete on a nonexistent bitfile ID.
- *new_bitfiles* - a directory containing *incomplete* bitfiles (i.e., in the process of creation).
- *bitfiles* - a directory hierarchy containing *immutable* bitfiles. Currently, a 3 tier hierarchy is used, with up to 256 entries per directory, permitting a maximum of 16M bitfiles per store.

Commands are provided to create, remove, relocate, and alter the configuration parameters of stores. Resource management is provided through a BSD-style quota mechanism that controls the total virtual storage, not just the magnetic disk storage. To keep quotas for each store independently, a private user ID is assigned to each store. This ID is used to set the owner of each file created in the store. The bitfile reference count is kept in the group ID of each bitfile.

RISP protocol service is provided by a collection of daemons. At the top level is *rssdad*, whose sole purpose is to make sure that its child process, *rssd*, is kept alive. The *rssd* process manages TCP connections from client systems. It advertises the RPC service and otherwise waits for connection requests and child terminations. When a connection arrives, it forks a child process referred to as a *client agent* to handle requests for that connection. Limits are enforced on the total number of outstanding connections and on the number of connections per client system. *Rssd* also controls the shutdown of RIS service when requested.

Each client agent works on behalf of a single client system for the lifetime of the agent. When a request arrives, the agent is responsible for authenticating the request and checking access rights to the stores being operated on. Shared memory segments are used to keep aggregate statistics among all agents and to maintain up-to-date bitfile index counters for ID generation within each store. Because multiple agents are allowed to access a single store at the same time, they must coordinate access to shared data structures. If a connection

stays idle for too long (10 minutes by default), the agent exits of its own accord. In addition, `rssd` will terminate agents if the configuration database is changed or the server is too busy.

RIS Client

The RIS client software implements Epoch-1 style InfiniteStorage management (ISM) using the RIS server as the mass storage vehicle. The Sun-based implementation of the RIS client consists of the following components:

- *filesystem wrappers* - a (usually) thin layer that is inserted between the VFS layer and the UNIX filesystem code during system boot.
- *ISA pseudo-driver* - a character device driver that provides kernel primitives for basic ISM operations and mediates communication between the filesystem wrappers and the ISM daemons.
- *ISM daemons* - a collection of daemons that handle file migration and other management functions for ISM.
- *user commands* - a collection of commands for configuring ISM, controlling staging of individual files, or replacing existing commands (such as `find` or `dump`) that require enhancements to work properly in a RIS environment.

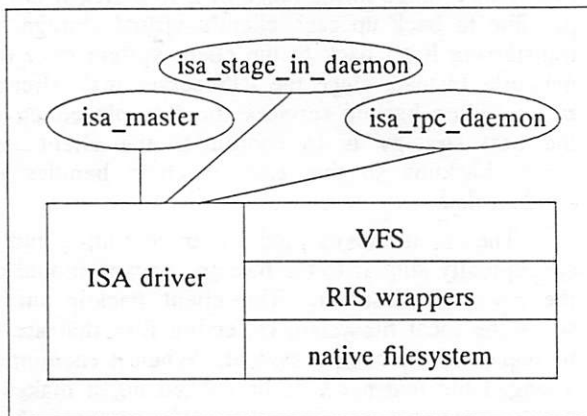


Figure 7: RIS Client Architecture

The RIS configuration database describes the local filesystems under ISM control and the client stores that belong to the client. Initially there are no filesystems under ISM control, and thus the filesystem wrappers pass all VFS operations directly through to the native filesystem. When a filesystem is configured for ISM, an `isa` directory is created on the filesystem. This directory holds an *extended attributes* file called `epxattr` and a `candidate_list` file (to be described below). The creation of these files activates the wrappers for that filesystem, initiating storage management.

The extended attributes file contains one extended attributes structure for each inode on the managed filesystem. The extended attributes

maintained by ISM include:

- the inode generator count, used to verify that the extended attributes entry is consistent with the corresponding inode.
- the *staging ID*, valid only for *regular* files and set only if the file is staged.
- the *staging priority*, a value that is used in conjunction with the file's access time, modification time, and size to rank the file as a candidate for stage-out.
- the *lock* flag, which when set prevents the file from being staged.
- the *stage when convenient* flag, which when set, causes the file to be staged during the next periodic staging run.

The latter three attributes can be set as *inheritable* attributes on a directory. Inheritable attributes are automatically set on any file or directory created beneath the given directory. Inheritance provides a convenient way to specify special staging properties for an entire file hierarchy.

To avoid unnecessary accesses of the `epxattr` file for magnetic-resident files, the filesystem wrappers maintain an in-memory bitmap for each managed filesystem that identifies files that are known to not be staged. When a file is to be accessed or modified the bitmap is consulted. If the corresponding bit is not set, the extended attributes are retrieved from the `epxattr` file. If the file is not staged, the bit is set in the bitmap and the operation is passed through to the native filesystem. All future accesses to this file thus avoid an access of the `epxattr` file.

The wrappers do little themselves except detect conditions that require ISM processing and pass notices on to the user level daemons that manage them. Stage-in and delete operations are handled by the `isa_stage_in_daemon` processes, several of which are started at boot time. Stage-out operations are handled by the `isa_master` process. The `isa_rpc_daemon` services network RPC requests to set extended attributes or stage-out individual files. RIS client commands that use the RPC service can be run both locally and from NFS clients of the managed filesystem.

Reading a non-resident staged file triggers the following sequence of events. A read access to a file enters the wrappers either via the `rdwr` (read/write) call or the `getpage` (page in for mapped files) call. The magnetic block count in the inode is zero, so the bitmap is checked to determine if the file is not staged. The bitmap indicates that the file *might* be staged, so the `epxattr` entry is read to find out. The staging ID is found to be non-null, so the wrapper posts a stage-in request to a `stage_in_daemon` via the `isa_driver` interface and goes to sleep. The `stage_in_daemon` reads the

stage-in request from the driver, consults the local configuration database to determine the server on which the indicated client store resides, and connects to the RIS daemon on that server. The client daemon then reads the data from the indicated bitfile and writes it to a temporary file on the same local filesystem as the original staged file. When the bitfile has been completely read, the stage_in_daemon makes a special `iocctl()` to the isa_driver that transfers the disk block map from the inode of the temporary file to the inode of the staged file and then awakens the process blocked on the stage-in. That process resumes execution in the wrapper function, which now calls the native filesystem `rdwr` or `getpage` operation to read the data. Note that once the read is done the file data resides both on magnetic storage and on the RIS server.

A stage-in is also required when a file is modified, either by a write or by truncation to a non-zero size. Once the data is magnetic resident, the staging ID in the extended attributes must be cleared and a bitfile delete operation forwarded to the RIS server. In the event that a staged file's data are completely discarded, as when the file is removed or truncated to zero size, the stage-in is skipped. To avoid a performance penalty for removing staged files, the bitfile delete operations are usually performed asynchronously.

The wrappers also detect high water mark crossings and out of space conditions. Each operation that may consume magnetic disk space checks for these conditions and notifies the isa_master process if they are present. If an operation is likely to result in an `ENOSPC` (out of space) error or such an error is actually returned from a native filesystem operation, the process is blocked until isa_master is able to free up magnetic space. Once sufficient space is available, the native filesystem operation is allowed to proceed.

User commands are provided to list and set the ISM attributes of files and to explicitly move files between magnetic disk and the RIS server. These commands actually work by making RPC calls to the isa_rpc_daemon and can be run from NFS clients of the managed filesystem or even on the NFS-mounted filesystems of an Epoch-1 fileserver. When multiple files are to be staged in together, the RIS client coordinates with the server to minimize the number of media exchanges in the server's jukeboxes.

RIS also provides replacements for several standard UNIX commands. The replacement for `du` provides an additional switch that reports on total virtual space allocated to files. By default `du` only reports on magnetic space allocation. The replacement for `tar` alters the extraction behavior to set the access time for files created to the modification time stored in the tar format. This provides better input to ISM's residence priority calculations. The `find` command has been augmented with two new

predicates, `-staged` and `-locked`, that are useful in writing shell scripts to perform custom management functions on ISM-controlled filesystems. Lastly, replacements for `dump` and `restore` are provided that work properly with the extended ISM information. The use of these commands in backing up RIS clients is described in the next section.

Administrative commands are provided to manipulate the RIS configuration information and to display statistics on filesystem usage and client store usage. The administrator can configure which filesystems are under ISM control, their water marks, the timing of periodic stage-out runs, and the stores to which files are staged out. Filesystem statistics include the total virtual and magnetic space devoted to files of various types, histograms of the number of files in buckets of sizes or age, and histograms of storage use against file size or age. Client store statistics include the space and number of bitfiles used on a store, the corresponding quota limits, and the amount of space or bitfiles actually available for further allocation.

Client Backup

On typical RIS clients at Epoch, the total virtual space consumed by clients is 5 to 10 times the actual magnetic space present. With many clients managing storage in the range of 2 to 5 GB, it is not possible to back up each client's virtual storage by transferring it all back to the client system over the network. Instead, since the RIS server may already be providing backup services for data placed on it, the best strategy is to coordinate the client and server backups so that each machine handles its resident data.

The client backup model is quite simple and is conceptually similar to the baseline backup model on the Epoch-1 fileserver. The client backup utility scans the local filesystem collecting files that are to be copied to the backup saveset. When it encounters a staged file that needs to be backed up, it makes a `bitfile_status` request to the RIS server, which returns an indication of the backup status for the bitfile on the server. If the bitfile has already been backed up on the server the client backup does not bother saving the data locally. If the bitfile was not backed up, or a magnetic copy of the data is present on the client, then the client backup makes a redundant copy of the data in the backup saveset. Thus when the client backup is completed, the client is guaranteed that all of its data is backed up either on the server or in the client's saveset.

The file recovery model is somewhat more complicated. When a staged file needs to be restored to a managed filesystem, the recovery utility notifies the RIS server of a new reference to the indicated bitfile using the `undelete_bitfiles` request. If the bitfile is present on the server, its reference

count is incremented. Otherwise the bitfile ID is added to the store's *recover_list* file and the client restore moves on. Periodically a utility called *rssundel* is run on the server that causes all bitfiles listed in *recover_list* files to be retrieved from the server's backup system. Only at this point is the staged file fully recovered. To help the client system determine when the data is available on the server, a command has been provided to poll the recovery status of individual bitfiles. The *RIS restore* utility uses this command internally to wait for full data availability before exiting.

Dump and Restore

Although Renaissance Backup will be the backup system of choice for RIS clients, ISM-compatible versions of the standard *dump* and *restore* utilities are also provided with RIS. This permits the use of RIS without requiring the development of new backup procedures. The existing dump saveset format is preserved by keeping all extended file attribute data in the *epxattr* file, which is always saved by the RIS dump. The RIS restore utility extracts the *epxattr* file internally to provide the extended attribute information for the remaining files in the dump saveset.

This design forced a minor alteration in ISM filesystem management. The dump format requires files to be saved in inode order, preventing the placement of the extended attributes file at the beginning of the dump saveset. Because restore needs this information to work correctly, two passes over the saveset would be needed to handle files that precede the attributes file in the saveset. Since making two passes over a multiple tape volume saveset is not acceptable, ISM was changed to prevent the stage-out of files whose inode number is less than that of the extended attributes file. Thus *restore* does not encounter files requiring extended attribute information until after the attributes have been extracted from the dump saveset. To limit the size of the resulting unstageable portion of the filesystem, the configuration software ensures that the attributes file is created with a low inode number.

Since the primary goal of RIS is to provide *automated* storage management, a shell script is provided that automates the dump process for client workstations by taking advantage of the vast storage facilities of the Epoch-1 fileserver. This script is run nightly via *cron*. It performs a dump on each local filesystem and writes the dump saveset to a private directory on the Epoch-1 server via NFS. The saveset is given a file name identifying the filesystem, dump level, and time of the dump. Options are provided to compress the dump saveset and to divide it into multiple files if it is excessively large. Since this procedure implies that the filesystems being dumped are not unmounted, there is some risk that an individual dump saveset will be fatally

inconsistent if the client system is not totally quiescent during the dump run. For installations that have predictable quiet times, the risk is relatively low.

Even for the Epoch-1 server the size of these dump savesets will eventually grow to uncomfortable proportions if left alone. Fortunately, over time the vast majority of dump savesets are redundant. The policy at Epoch is to keep the last 4 savesets at each dump level for each client filesystem and to delete the rest. All saveset files are backed up to tape so that a given saveset can still be retrieved by the recovery system after it has been deleted from online storage.

Evaluation

The combination of RIS with the Epoch-1 fileserver has been quite successful in reducing the administrative effort required for network storage management while improving the working environment for workstation users. After initial installation and configuration, storage management proceeds automatically and is usually unnoticed by users. Overall performance and accessibility of data are improved for end users because their working set of data is kept entirely on their local disk. At Epoch, RIS users are often completely unaware of times when the server or network are down.

operation	non-ISM	cold	warm
create	18 files/s	6 files/s	7 files/s
stat	544 files/s	484 files/s	500 files/s
read	383 KB/s	298 KB/s	305 KB/s
write	347 KB/s	325 KB/s	330 KB/s
remove	54 files/s	51 files/s	54 files/s

Figure 8: Wrapper Overhead

Performance

There are three aspects of performance that are critical to RIS: degradation of local filesystem access due to the wrapper overhead; local magnetic "cache" hit ratio; and the transfer rates for stage-in operations. The performance of stage-out operations is generally less important, since stage-out occurs mostly at off hours and asynchronous to client activity.

The measurements of filesystem wrapper overhead test three cases: access to unmanaged filesystems; access to "cold" (newly mounted) filesystems; and access to "warm" filesystems, all with files magnetic-resident. The primary difference between the latter two cases is that in cold filesystems the

bitmaps for extended attribute handling have not been initialized.

Direct measurements of Epoch-1 RIS server performance show that the server is capable of handling stage-out rates in excess of 300 KB/second and stage-in rates in excess of 250 KB/second, with sustained rates of 4 bitfile creations per second and 12 deletions per second. Unfortunately, the staging algorithms used in the initial RIS client implementation do not achieve these raw transfer rates. The staging performance numbers for a Sparcstation 1+ RIS client are:

operation	performance
stage-in (files)	5 files/s
stage-in (data)	126 KB/s
stage-out (files)	1.8 files/s
stage-out (data)	182 KB/s
remove	6.3 files/s

Storage Distribution

There are two measures of storage usage that are critical to RIS: the client's working set size and the overall virtual to physical storage ratio. For applications whose working set size does not depend on the total size of managed storage, the virtual to physical ratio can be practically unlimited. These applications are referred to as *archival* applications, because data outside of the working set are rarely accessed so the delay involved in retrieving them is not important. An example of such an application in the Epoch engineering environment is the software release tracking system, in which a complete snapshot of the source code for every software release ever produced is kept.

Other applications have a working set that tends to grow with the overall amount of data. This is certainly true in software development, where the complete source code for active products is always required. In extreme cases the working set of storage is the total virtual storage, in which case the only viable solution is to keep the data physically resident on the client.

In general, however, network environments have a range of applications with varying working set requirements. This can blur the boundaries of the working set and place great importance on the algorithm used to select candidates for stage-out. The current LRU policy for ranking files has been very successful; other policies based on locality or ownership may be beneficial in complex cases. Another approach to mixed applications is multi-

level staging, in which data are initially staged to a relatively accessible medium based on LRU, and later restaged to an archival medium as the intermediate storage fills.

Since existing applications have been developed in an environment with very primitive storage management, most of them present a well-defined working set with a well-defined transition to archival storage. As a general rule, RIS can support a virtual to physical storage ratio of at least 5 to 1 in most environments.

Operation

RIS is designed to make day-to-day use of storage worry-free and it has been very successful in this. For the most part users of RIS client systems are unaware of its presence. The presence of RIS becomes apparent mainly through the absence of "file system full" failures and the delays encountered when the user wanders outside of the storage working set.

For the facility administrator, some initial effort in designing and configuring the storage management strategy results in significantly reduced long term effort in daily management tasks. Using the Epoch-1 RIS server, the daily management tasks consist mostly of media management for backup and ISM. This generally takes only a few minutes and can be done at the administrator's convenience rather than according to rigid time schedules.

Although RIS eliminates hard limits on storage usage, users will eventually encounter softer limits as their working sets begin to exceed the local magnetic storage. The statistics gathering tools provided by RIS will help the network administrator identify developing problems before they become apparent to the users. Thus, facility planning is improved and service disruptions for users are minimized.

Future Directions

Epoch Systems is currently working to extend the functionality and scope of its storage management products. Current areas of investigation and development include:

- tools for centralized configuration, control, and monitoring of RIS clients and servers.
- improved user and administrator interfaces.
- improvement of the ISM staging algorithms to take into account locality properties such as file directory or ownership.
- use of ISM in other storage paradigms, such as relational databases or AFS volumes.
- integration into alternative networking environments such as the OSF DCE or OSI.

With the ongoing explosion of data being generated and accessed, automated storage management will become an essential aspect of networked computing environments. Epoch's InfiniteStorage Architecture is one approach to managing storage that matches well with the current structure of networks and applications. The concepts and techniques developed by Epoch will play an important role in the future as networked applications integrate more closely with data management facilities.

Acknowledgements

Many of the concepts presented in this paper were developed by other members of Epoch Systems' engineering group, including George Ericson, Rich Fortier, Steve Glassman, Mark Hecker, Ken Holberger, Chuck Holland, Greg Kenley, Rob Kenna, Sandy Laird, Dave Noveck, Bob O'Donoghue, Jim Pownell, Dave Therrien, and John Wallace. A special thanks to Laura Israel and Dianne Pearson for their editorial assistance.

References

- [Arneson88] David A. Arneson, "Mass Storage Archiving in a Network Environment", *Digest of Papers, Ninth IEEE Symposium on Mass Storage Systems*, IEEE Catalog Number 88CH2626-0, October, 1988.
- [Howard88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems*, Volume 6, Number 1, February 1988.
- [IEEE90] IEEE Technical Committee on Mass Storage Systems and Technology, *Mass Storage Systems Reference Model: Version 4*, May, 1990.
- [Israel89] Robert K. Israel, Sandra Jett, James Pownell, and George M. Ericson, "Eliminating Data Copies in UNIX-based NFS Servers", *Uniform 1989 Conference Proceedings*, March 1989.
- [Kenley90] Gregory G. Kenley, "An Architecture for a Transparent Networked Mass Storage System", *Digest of Papers, Tenth IEEE Symposium on Mass Storage Systems*, IEEE Catalog No. 90CH2844-9, May, 1990.
- [Kleiman86] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *USENIX Summer Conference Proceedings*, 1986.
- [Morris86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith, "ANDREW: A Distributed Personal Computing Environment", *Communications of the ACM*, Volume 29, Number 3, March 1986.
- [Muuss88] Michael John Muuss, Terry Slattery, and Donald F. Merritt, "BUMP, The BRL/USNA Migration Project", *Unix and Supercomputers*, 1988.
- [RPC86] Sun Microsystems, Inc., *Remote Procedure Call Protocol Specification*, 1986.
- [Sun86] Sun Microsystems, Inc., *Network File System Protocol Specification*, 1986.
- [Taylor89] T. M. Taylor and R. W. Fortier, "Using Optical Disks to Extend the Capacity of Magnetic Disks Through Hierarchical Storage", *Uniform 1989 Conference Proceedings*, March 1989.
- [XDR86] Sun Microsystems, Inc., *External Data Representation Protocol Specification*, 1986.

Robert K. Israel is a Principal Software Engineer at Epoch Systems, Inc. His recent work includes the design and implementation of the Renaissance InfiniteStorage Manager and of NFS for the Epoch-1 fileserver. Previously, he worked on TCP/IP at Micom-Interlan and STREAMS at AT&T. Mr. Israel holds a M.S. in Computer Science from Washington University. Reach him via U.S. Mail at Epoch Systems, Inc., 8 Technology Drive, Westborough, MA 01581. His electronic mail address is rki@epoch.COM.



Antony Foster is a Senior Engineer at Epoch Systems, Inc., working on the Renaissance InfiniteStorage Manager. Previously at Data General he was responsible for Open Network Computing applications (such as NFS) on the AOS/VS II platform. He has a B.S. degree in Computer Engineering from Case Western Reserve University. Reach him via U.S. Mail at Epoch Systems, Inc., 8 Technology Drive, Westborough, MA 01581. His electronic mail address is afoster@epoch.COM.



Arun Taylor is a Principal Software Engineer at Epoch Systems, Inc., where he works on porting the Renaissance InfiniteStorage Manager to a variety of workstation and fileserver platforms. Previously at Sun Microsystems he worked on OpenLook/OpenWindows and ONC based Distributed Applications. He holds a B.Sc(Engineering) from Imperial College of Science, Technology and Medicine, London, UK. Reach him via U.S. Mail at Epoch Systems, Inc., 8 Technology Drive, Westborough, MA 01581. His electronic mail address is arun@epoch.COM.



Tracy M. Taylor is a Principal Engineer at Epoch Systems, Inc., where he is involved with various aspects of the InfiniteStorage Architecture development. Previously, he worked on a high-performance, fault tolerant multiprocessor filesystem implementation at Sequoia and on distributed filesystems at Prime. Reach him via U.S. Mail at Epoch Systems, Inc., 8 Technology Drive, Westborough, MA 01581. His electronic mail address is tmt@epoch.COM.



Neil Webber is a Principal Software Engineer at Epoch Systems, Inc., working on the Renaissance InfiniteStorage Client implementation. His earlier work includes a port of BSD 4.3 to the Epoch-1 platform, and the development of a proprietary real-time multiprocessor operating system at Automatix Inc. He has a B.S. in Computer Science from MIT. Reach him via U.S. Mail at Epoch Systems, Inc., 8 Technology Drive, Westborough, MA 01581. His electronic mail address is nw@epoch.COM.



A Highly Available Network File Server

Anupam Bhide – IBM T.J. Watson Research Center
Elmootazbellah N. Elnozahy – Rice University
Stephen P. Morgan – IBM T.J. Watson Research Center

ABSTRACT

This paper presents the design and implementation of a Highly Available Network File Server (HA-NFS). We separate the problem of network file server reliability into three different subproblems: server reliability, disk reliability, and network reliability. HA-NFS offers a different solution for each: dual-ported disks and impersonation are used to provide server reliability, disk mirroring can be used to provide disk reliability, and optional network replication can be used to provide network reliability. The implementation shows that HA-NFS provides high availability without the excessive resource overhead or the performance degradation that characterize traditional replication methods. Ongoing operations are not aborted during fail-over and recovery is completely transparent to applications. HA-NFS adheres to the NFS protocol standard and can be used by existing NFS clients *without* modification.

Introduction

Traditional approaches for providing reliability in network file systems by server replication suffer from excessive resource overheads, performance degradation, and increased complexity. Replicated servers use expensive protocols to maintain consistency and coherence, leading to performance degradation during failure-free operation. They also use complex protocols to update the state of a stale replica when it is repaired after failure. Further, handling network partition requires quorum management, increasing system complexity.

This paper describes the design and implementation of a Highly Available Network File Server (HA-NFS) that adheres to the semantics of SUN's Network File System (NFS) [1]. HA-NFS differs from traditional approaches in that it considers the problem of providing a reliable network file system as three separate subproblems, namely: recovering from server failures, recovering from disk failures, and recovering from network failures. HA-NFS offers a different solution to each of these subproblems.

Server failures are tolerated by using dual-ported disks that are accessible to two servers, each acting as a backup for the other. The disks are divided into two sets, each served by one server during normal operation. Each server maintains on its disks enough information to reconstruct its current volatile state. The two servers periodically exchange liveness-checking messages. If one server fails, its disks will be taken over by the other server, which will reconstruct the lost volatile state using the information on disk. Then, it *impersonates* the failed server, and operation continues with a potential reduction in performance due to the increased load. The machines on the network are oblivious to the

failure, and continue to access the file system using the same address. During normal operation, the servers communicate only for periodic liveness-checking. The servers do not maintain any information about each other's volatile state or attempt to access each other's disks.

Fast recovery from disk failures is achieved by mirroring files on different disks. However, all copies of the same file are on disks that are controlled by the same file server, eliminating the overhead of ensuring consistency and coherence between the two servers that would otherwise occur. Since disk failures are not frequent, mirroring is only used for applications that require continuous availability. Otherwise, archival backups could be used to recover from disk failures.

Network failures are tolerated by optional replication of the network components, including the transmission medium. However, packets are not replicated over the two networks. Instead, the network load is distributed over the networks.

HA-NFS servers conform with the server protocol of SUN's NFS. NFS has gained wide acceptance as a general purpose network file system. By adhering to a standard file system, our results can have direct application in practical environments. In addition to adherence to standards, our design has several important goals:

- * Failure and recovery must be completely transparent to applications running on the file-server's clients. A failure must not force operations in progress to terminate.
- * Failure-free performance must not be penalized to provide high availability.
- * NFS client protocol implementations should not require modification to use HA-NFS servers.

We have implemented a prototype of HA-NFS on a network of workstations and two file servers from the IBM RISC System/6000 family of computing systems running the AIX Version 3 (AIXv3) operating system, and connected by both a 10 Mbit/s Ethernet network and a 4 Mbit/s token ring network. We construct dual-ported disks from off-the-shelf SCSI disks attached to a SCSI bus that is shared by the two servers. The prototype is operational and has satisfied the design goals.

In section "Background", we present background information on NFS and AIXv3. We present the design of an HA-NFS server in section "HA-NFS Architecture". We discuss the performance in section "Performance". We compare our design to related work in section "Related Work". Finally, we draw conclusions and outline future work in section "Conclusions and Future Work".

Background

HA-NFS is implemented on top of the AIXv3 journaled file system. The AIXv3 file system provides serializable and atomic modification of file system meta-data by using transactional locking and logging techniques. File system meta-data are composed of directories, inodes, and indirect blocks. Every AIXv3 system call that modifies the meta-data does so as a transaction, locking meta-data as they are referenced, and recording the changes in a disk log before allowing the meta-data to be written to their "home" locations on disk. In the case of system failure, the meta-data are restored to a consistent state by applying the changes contained in the log. The reliability of ordinary files is ensured by NFS semantics, which require forcing the data to disk before sending an acknowledgement to the client.

AIXv3 supports *logical volumes*, which provide the abstraction of logical disks. Logical volumes can be mirrored to provide disk reliability. Each logical volume can have up to three copies, each on a different physical disk.

Although NFS is defined as a stateless file server protocol, most NFS implementations maintain a small amount of state information. Some NFS operations, such as erasing a file, cannot be implemented by idempotent remote procedure calls (RPC). An NFS server maintains a *reply cache*¹ unsuccessful, the server uses the reply cache to tell whether the RPC is a retry of a previously successful, non-idempotent RPC. If it is, the server responds to the client that the RPC completed successfully. HA-NFS records changes to this volatile state in the AIXv3 disk log, so that the reply cache can be reconstructed in the case of failure.

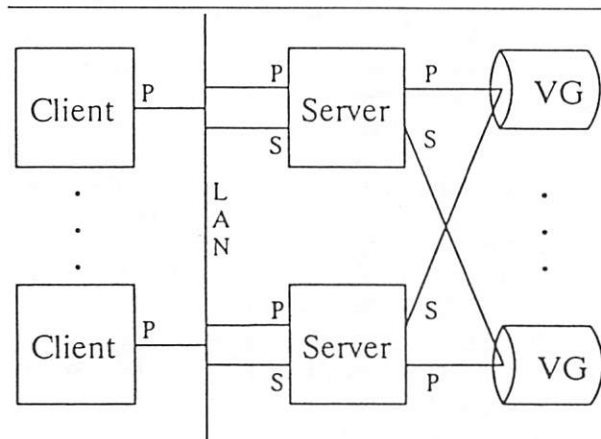
¹Also called a duplicate cache.

HA-NFS Architecture

An HA-NFS node consists of two NFS servers sharing a number of SCSI buses. Each shared SCSI bus and the disks connected to it have one of the servers designated as their *primary server*. During normal operation, the disks are served only by their corresponding primary server, the other server does not access the bus. The primary server for each bus is selected such that the total load is balanced (statically) over the two servers. Both servers act as backups for each other. NFS clients perceive an HA-NFS node as two independent NFS servers, each serving a distinct set of file systems.

Each server has two network interfaces and IP addresses. The server uses its *primary interface* for normal operation, and its *secondary interface* when *impersonating* the other server after its failure. The server also uses its secondary interface when re-integrating with the system after repair or maintenance.

Figure 1 shows a single HA-NFS node consisting of two servers on a single network.



Key: P: Primary adapter
S: Secondary adapter
VG: Volume group

Figure 1: Two servers on one network

Normal Operation

During normal operation, a server performs the operation described in each NFS RPC it receives. If the operation is successful, the server will record the meta-data changes in the AIXv3 file system log and enough information to identify the RPC if it is non-idempotent. (This information is identical to that in the volatile reply cache and will be used in the case of failure to reconstruct the volatile state.) If the operation completes successfully and is non-idempotent, the server will add an entry in its reply cache for the RPC.

If the operation did not complete successfully, the server will determine whether there is an entry in the reply cache corresponding to the RPC. If an entry is found, then the RPC is a retry of a non-idempotent operation that succeeded before. The server will reply to the client that the RPC completed successfully; otherwise, the server replies to the client with an appropriate error code, indicating the failure of the requested operation.

Both servers in the node exchange NFS RFS_NULL RPC's to monitor the liveness of each other. An RFS_NULL is a "no operation" RPC that is echoed back from the server if it is running. If a server does not receive an acknowledgement for the RFS_NULL after a specified number of such RPC's, it will start failure detection. First, it checks if it can "ping" the suspect server by sending an ICMP echo packet. Second, it attempts to communicate with the suspect server via the shared SCSI bus in "target mode". This is analogous to pinging except that the requests are sent over the SCSI bus rather than the network, and the response is sent by a device driver which must respond within a certain period of time. Both the ICMP communication and the target mode communication on the SCSI bus are performed conceptually at the interrupt handler level. Thus, a response is generated even though the server may be so overloaded that it cannot respond to NFS RPC's.

If a response is obtained from either of these two tests, it is likely that the suspect server is undergoing a period of slow response. The failure detection tests are conservative, so it is possible that the tests indicate that a server is alive while it is "brain-dead", i.e., able to respond correctly to the tests, but incapable of processing NFS RPC's. In such unlikely cases, HA-NFS refrains from continuing the take-over and relies on operator intervention. In a network, it is impossible to determine with absolute certainty whether a certain machine has failed [2]. However, the failure detection tests never declare a server dead while it is operational. This prevents a race condition where both servers attempt to access all the disks in the node at the same time, which can lead to corruption of the file systems.

Take-over

If a server fails, its disks will be taken over by the other server. The live server brings the failed server's volume groups on-line by running their logs and restoring the file systems to a consistent state. The server also uses the log to retrieve the reply cache entries of the failed server and inserts them in its own cache. Then, it starts impersonating the failed server by changing the IP address of its secondary network interface to the primary address of the failed server. The live server also changes the hardware address of its secondary interface to that of the primary interface of the failed server. Thus,

packets that were intended for the failed server can now be received by the live server on its secondary interface.

If network interfaces that can change their hardware address are not available, an alternate scheme may be used to allow the live server to receive the packets intended for the failed one. The scheme consists of using the ARP [3] protocol to update the mapping between the failed server's IP address and the hardware address to reflect the change. HA-NFS updates stale mappings in the clients' ARP caches by sending an ARP request which queries for the hardware address corresponding to some machine's IP address. The query appears to have been sent from the failed server's IP address, but with the live server's secondary interface as the source hardware address. On receiving this ARP broadcast, each machine on the local network updates its mappings to reflect the change. The update is automatically performed by the ARP protocol layer on the clients, so no modification in the network software is necessary. The broadcast is repeated several times to ensure that virtually all clients on the local network will eventually receive it.

We have decided to use the hardware address change approach in our final prototype since the ARP approach relies on the correct implementation of the ARP protocol on all types of clients. However, the take-over time reported in section 4 was measured using the ARP approach since at that time we did not have the special type of network interfaces that allowed changing hardware addresses.

During take-over, clients of the failed server continue to retransmit their requests. When the live server starts to impersonate the failed one, it receives the clients' requests and begins to serve them. Clients are oblivious to the change, all they can detect is that the server has gone through a period of slow response.

Re-Integration

When a server comes up, either normally or after repair or maintenance, it cannot immediately configure its primary network interface to its primary IP address, since it may be impersonated by its backup.

Instead, the server comes up with its primary network interface turned off, and uses its secondary interface to send a re-integration request to the backup.

If the backup is running, it will acknowledge the receipt of the re-integration request. After unmounting the corresponding file systems, the backup switches the IP and hardware addresses of its secondary network interface back to their normal settings, thereby stopping the impersonation of the other server. Finally, the backup sends a message to

the re-integrating server allowing it to proceed. The re-integrating server reclaims its SCSI buses and disks, runs the log and reconstructs the reply cache, switches its own primary interface on, and starts serving NFS requests.

Care is taken to recover from failures of either server during re-integration. The servers periodically exchange liveness messages until the backup relinquishes the buses. Communication through the SCSI buses will also be used if either server suspects the failure of the other. If the re-integrating server fails, the backup will reclaim the disks as in take-over. If the backup fails, the re-integrating server will start normal operation on its own. Later, it will start impersonating the failed backup.

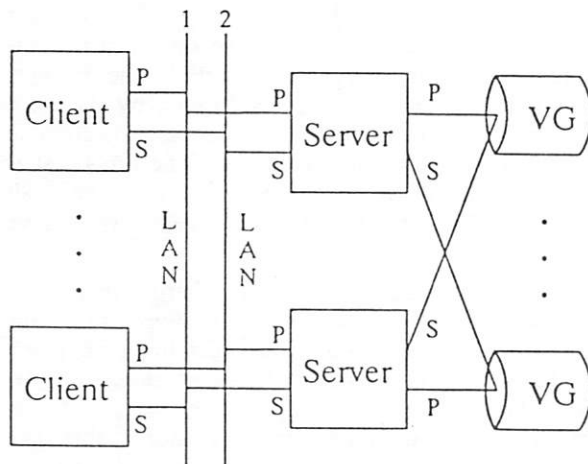


Figure 2: Two network configuration

Network Failure

To tolerate network failures, HA-NFS relies on replicating the network. Figure 2 shows an HA-NFS node in a two-network configuration.

Recovery from server failures does not require any changes to clients. Recovery from network failure, however, requires a daemon to run on the client to observe the status of each network and reroute requests to the operational network if a failure occurs. Since the daemon is run as a user process, no change to the kernel or to the NFS protocol is necessary.

When an HA-NFS node is connected to two networks, each server has its network interfaces connected to different networks. Further, the two servers have their primary interfaces on different networks. Thus, the servers receive their requests on different networks, which provides a degree of network (static) load balancing. Clients are also configured to (statically) balance the load on both networks.

In addition to its roles during take-over and re-integration, the secondary network interface now serves as an alternate path to the server, should its primary interface become unreachable because of a network failure. Each server broadcasts a "heartbeat" message from its primary interface. The daemon on every client detects the heartbeats of the servers on both networks. When the daemon detects the loss of the heartbeat of one server, the daemon concludes after a timeout period that the path to the server's primary network interface is broken. In this case, the daemon updates the client's routing table to use the alternate path to the server. The daemon also sends a request to the daemon on the server to update the routes for RPC acknowledgements to that client to the operational network, if necessary². Once the daemon detects the return of the server heartbeat on a broken path, it restores routing to its normal setting and requests that the server reroute the path for the RPC acknowledgements, if necessary.

When a server takes over the role of its counterpart in the HA-NFS node, it needs to broadcast a heartbeat on behalf of the failed server, so that clients continue to believe that the server is still reachable across its default path. The server will set its secondary interface's IP address to that of the failed server's primary interface (which is on the same network). Combinations of network and server failures are tolerated. For example, a server taking over the role of a failed server may face a failure on the network on which the failed server's primary interface resides. In this case, the daemons on the clients should route requests for the failed server to the primary interface of the live server, since the secondary interface used for impersonation is now unreachable.

Performance

HA-NFS provides high availability without incurring excessive performance penalty. We measured the performance of HA-NFS by running a set of experiments on a number of RISC System/6000 family workstations (25 MHz), connected by a 10 Mbit/sec Ethernet. All measurements were obtained by directly calling the SUN RPC layer, bypassing the NFS client cache. The underlying system uses 4 KByte disk blocks.

The Effect of Disk Logging

Table 1 shows a comparison between HA-NFS, and a traditional implementation of NFS that does not use disk logging. The traditional implementation of NFS forces the data and the meta-data to their

²If the client's main address is on the operational network, then the server need not reroute the acknowledgements.

home locations on disk before responding to the RPC. In contrast, HA-NFS records meta-data modifications as a log record, requiring no disk arm movement. The meta-data are written asynchronously to their home locations. Because the reply cache entries are piggybacked on the normal disk log information, saving the volatile state on disk does not incur appreciable overhead beyond the cost of basic disk logging. As expected, disk logging improves the response time of all RPC's that modify the file system structure, such as SETATTR, CREATE, REMOVE, MKDIR, and RMDIR RPC's. The table shows that disk logging improvement ranges from 33% for SETATTR and WRITE RPC's, up to 75% for MKDIR RPC.

In the above measurements, the log was placed on a separate disk. Placing the disk log on the same disk with the data reduces the performance gain due to the extra disk arm movement. For example, the improvement in CREATE RPC drops from 58% to 20%.

	HA-NFS	NFS	Improvement
	(ms)	(ms)	%
NULL	5.26	5.26	0
GETATTR	6.04	6.04	0
SETATTR	32.08	48.32	33
LOOKUP	6.96	6.96	0
READ	12.13	12.13	0
WRITE	72.28	108.80	33
CREATE	38.07	91.81	58
REMOVE	35.22	87.37	60
RENAME	35.58	75.96	53
MKDIR	37.20	151.47	75
RMDIR	34.73	118.40	70
REaddir	11.08	11.08	0
STATFS	6.05	6.05	0

Table 1: Traditional NFS vs. HA-NFS.

The Effect of Mirroring

The only overhead introduced by mirroring is a 17% slow-down for the WRITE RPC. This overhead is attributed to the variation in the disk arm position among the mirrors at the time of writing to disk and to the performance overhead of the mirroring software. When client caching is turned on, the overhead drops to 2% at the application-program level.

Because of disk logging, mirroring does not introduce any overhead to the RPC's that maintain the file system structure (e.g., CREATE).

Take-over and Re-integration

A second set of measurements shows how long it takes a backup to take over the failed server's role, and how long it takes a recovering server to re-integrate with the rest of the system.

Failure detection consists of an empirically chosen timeout period of 10 seconds and a number of precautionary tests for liveness which take 5 seconds (these parameters are configurable.). We measured a total time of 15 seconds for the backup to perform all take-over operations, excluding failure detection. Thus, the backup takes about 30 seconds before starting to serve the disks of the failed server. During that time, the disks are not available.

We measured a total time of 60 seconds for a server to re-integrate into the system after repair. Re-integration takes longer than fail-over because the backup must wait for ongoing NFS RPC's to terminate in addition to the overhead of unmounting the corresponding file systems.

Related Work

HA-NFS is unique in that it considers the problem of providing reliability to file servers as three separate subproblems, namely: server reliability, disk reliability, and network reliability. Each subproblem is handled differently. Logging the server's volatile state to a disk accessible to the backup tolerates server failures, optional replication of disks tolerates disk failures, and optional replication of the network components tolerates network failures.

HA-NFS does not suffer from the problems associated with the traditional approaches based on replicating the file server as a unit [4] [5] [6] [7] [8] [9]. Replicating the file server as a unit introduces additional overhead during failure-free operation due to the need to enforce consistency among the replicas. Re-integrating a recovering server into the system can be expensive since it requires updating the server's stale view of the replicated file system. To tolerate network partition, a replication-based system must support read and write quorums, incurring a substantial performance penalty. This penalty has led some systems [4] [9] to abandon quorums, allowing divergence in replicas during network partition. While this solution may be acceptable in many practical environments, it cannot be relied on in general and it exposes failures to the users. In HA-NFS, we direct our effort to making the network more reliable independent of the solution we employ to provide server reliability. After all, the effects of a network partition are not limited to file servers. Also, the availability of a replicated file server is greatly compromised for the clients that are in a partition without a replica.

On the other hand, replicated file servers can distribute the file-read load on the different replicas to achieve load balancing. Replicated file servers

are also better suited for wide area networks where a client can access its files from the nearest replica, reducing network load during file read. A recent comparison study with the Deceit file server [10] shows that the performance benefits of HA-NFS and its relative simplicity come at the expense of a lack of flexibility. Both servers of an HA-NFS node must be physically close to each other because of the restriction on the SCSI bus length. HA-NFS cannot resist site disasters or total site failures. Installing an HA-NFS node is complicated by the provisions taken to ensure independent failure modes of the servers and the disks in the node.

Like HA-NFS, several reliable file servers attempt to provide reliability to NFS without changing the client implementation of the protocol [5] [6] [8]. HA-NFS is unique in that it uses impersonation to mask fail-over from the clients. In the other systems [5] [6], the clients continue to attempt to access the files from the failed server, therefore "hanging" until the user intervenes and remounts the file systems from an alternate source. The reliable file system of MIT [8] suggests the use of IP multicast addressing to solve this problem, but no implementation has been reported. When compared to impersonation, IP multicast increases the load on the replicas and introduces complexity, since all replicas must process the multicasts in the same order.

Using dual-ported disks is also the basis of the reliable file system of Tolerant [11] and the Echo [12] reliable file system. Tolerant and HA-NFS are similar in that they rely on a non-dedicated backup to provide reliability against server failure. However, Tolerant relies on transaction semantics at the application level, and ongoing transactions are aborted during fail-over. In contrast, HA-NFS does not rely on transaction semantics at the application level, and ongoing operations are not affected during fail-over. HA-NFS differs from Echo in that an HA-NFS backup does not maintain information about the current volatile state of the main server, and that HA-NFS clients are oblivious to the backup take-over. In Echo, the primary informs the backup about its state, and each client has a "clerk" layer that isolates the application programs from failures and recoveries.

Conclusions and Future Work

As modern computer hardware becomes intrinsically more reliable, traditional solutions to provide reliability in network file systems by server replication become less attractive because of the performance penalty and the complexity they incur. We have presented HA-NFS, a reliable file server based on offering different treatment to the reliability of each component of the file system. Our approach offers server reliability by using dual-ported disks and impersonation, disk reliability by using mirroring, and network reliability by replication. HA-NFS

is one of the few designs that recognize that network failures require an independent solution from that used to provide reliability to the file server.

Comparing the performance of HA-NFS with traditional implementation of NFS shows that disk logging improves the performance of many NFS RPC's. Mirroring, when used, adds only 17% overhead to WRITE RPC. Saving the reply cache entries is piggybacked on the normal disk logging operation, thus adding no more overhead beyond that of the basic disk logging.

Recovery in HA-NFS is completely transparent to applications and does not involve aborting ongoing operations. Impersonation prevents the client from "hanging" during fail-over. User programs and NFS client protocol implementation need no modification to use HA-NFS. HA-NFS shows that it is possible to add transparent fault-tolerance to existing systems without adding significant overhead.

The high performance and relative simplicity of HA-NFS come at the expense of a loss in flexibility. HA-NFS cannot tolerate more than one server failure. During fail-over, the disks are not available for a period of 30 seconds. The servers must be physically close because of the restriction on the length of the SCSI bus. HA-NFS cannot tolerate total site failures or site disasters.

We are currently addressing the shortcomings of HA-NFS. We are considering the use of optical links instead of the shared SCSI bus to simulate dual-ported disks. Because of their high performance and capability of extending over relatively long distances, optical links can remove the restrictions of installing servers and their disks in close proximity. This will also facilitate the realization of independent failure conditions and allow more than two servers to share the same disk. We are considering adding stable semiconductor memory on the disk controller to remove all the overhead of disk logging. We are also considering adding extensions to HA-NFS operations to support consistency of concurrent file access in the presence of client caching. Finally, we plan to use the HA-NFS methodology to provide higher availability for stateful server protocols such as Andrew [13].

References

- [1] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March, 1989.
- [2] M.J. Fisher, N.A. Lynch, and M.S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, 32(2):374-382, April 1985.
- [3] D. C. Plummer. Ethernet address resolution protocol; RFC826. In *ARPANET Working Group Requests for Comments*, no. 826. SRI

- International, Menlo Park, California, November 1982.
- [4] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Seigel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447-459, April 1990.
 - [5] Keith Marzullo and Frank Schmuck. Supplying high availability with a standard network file system. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 447-453, May 1988.
 - [6] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. Technical Report TR 89-1042, Cornell University, November 1989.
 - [7] Uppaluru Premkumar, W. Kevin Wilkinson, and Hikyu Lee. Reliable servers in the JASMIN distributed system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 105-112, September 1987.
 - [8] Barbara Liskov, R. Gruber, P. Johnson, and L. Shrira. A replicated Unix file system. In *Proceedings of the First IEEE Workshop on Management of Replicated Data*, pages 11-14, November 1990.
 - [9] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63-71, USENIX, June 1990.
 - [10] Anupam Bhide, Elmootazbellah N. Elnozahy, Stephen Morgan, and Alex Siegel. A comparison between two reliable file servers. In preparation.
 - [11] Dale L. Shipley, Joan D. Arnett, William A. Arnett, Steven D. Baumel, Anil Bhavnani, Chuenpu J. Chou, David L. Nelson, Maty Soha, and David H. Yamada. Distributed multiprocess transaction processing system and method. U.S. Patent No. 4819159, April 1989.
 - [12] Andy Hisgen, Andrew Birrell, Timothy Mann, Michael Schroeder, and Garret Swart. Availability and consistency tradeoffs in the ECHO distributed file system. In *Proceedings of the 2nd Workshop on Workstation Operating Systems*, pages 49-54, 1989.
 - [13] John H. Howard, Micheal L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, and Robert N. Sidebotham. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, pages 51-81, February 1988.

Anupam K. Bhide is a researcher working at IBM's T. J. Watson Research Centre. His research interests include database systems, operating systems, distributed systems and fault-tolerance. He graduated from I.I.T.-Bombay with a B.Tech. in Computer Science in May 1983. He obtained an M.S. in Computer Science from the University of Wisconsin-Madison in August, 1984 and a Ph.D. in Computer Science from University of California-Berkeley in October, 1988. Reach him electronically at ANUPAM@YKTVMH2.Bitnet.

Elmootazbellah N. Elnozahy is a Ph.D. candidate in the department of Computer Science, Rice University. His research there has been supported in part by an IBM fellowship. His research interests include operating systems, distributed systems, fault-tolerance, communications and parallel processing. He graduated from Cairo University, Egypt, with a B.Sc. in Electrical Engineering with Highest Honours, July 1984. He also obtained an M.S. in Computer Engineering from Cairo University, May 1987, and an M.S. in Computer Science from Rice University, May 1990. Reach him electronically at mootaz@rice.edu.

Stephen P. Morgan is an advisory programmer with the IBM T.J. Watson Research Center, where he has worked for nearly ten years. He received an S.B. degree in Computer Science and Engineering from MIT, February 1982. While on assignment to the IBM Advanced Workstations Division in Austin, Texas, from mid 1986 through mid 1988, Mr. Morgan helped architect various portions of the AIXv3 operating system, including the Logical Volume Manager. He was instrumental in forming the Open Software Foundation in 1988. His research interests range from distributed operating systems and database systems to neural networks and the development of software from formal specifications.

The OSF/1 Unix Filesystem (UFS)

*Susan LoVerso, Noemi Paciorek, Alan Langerman – Encore Computer Corporation
George Feinberg – Open Software Foundation*

ABSTRACT

The OSF/1 Unix File System (UFS) originated from the Berkeley 4.3-Reno distribution local filesystem code combined with parallelization modifications by Encore Computer Corporation. The Berkeley project concentrated exclusively on a uniprocessor implementation while previous Encore projects focused only on multiprocessor implementations. OSF/1, on the other hand, must run efficiently in both environments.

This paper presents an overview of the parallelized OSF/1 Unix filesystem and describes the rationale behind the changes we made. We discuss the addition of timestamps to optimize the single-stream performance of important, parallelized UFS algorithms. We also describe several interesting race conditions resulting from our new UFS locking protocols and the introduction of timestamps.

1. Introduction

OSF/1 derives from CMU Mach Release 2.5, 4.3BSD-Reno, Encore Mach/0.6, and sources from several other organizations. Mach has been described extensively elsewhere [9] [12]; it provides five basic abstractions: tasks and threads, memory objects, and ports and messages. These abstractions were designed and implemented to execute efficiently on shared memory multiprocessors as well as on uniprocessors.

In addition, Mach emulates a 4.3BSD operating system by incorporating Unix compatibility code from the 4.3BSD release. As distributed by Carnegie-Mellon University, Mach executes all of this code in a master/slave paradigm so that it continues to function as on a uniprocessor.

Encore parallelized the 4.3BSD compatibility code for the Multimax shared-memory multiprocessor. Parallelizing Unix is not new; the parallelization of Unix kernel software for shared memory multiprocessors has received increasing attention over the last few years. As early as 1984, AT&T Unix System Organization released a version of System V developed on a multiprocessor [1]. Shortly thereafter, Encore [3] and Sequent [10] released parallelized versions of 4.2BSD. More recently, the world has seen the introduction of parallelized Unix operating systems from many companies, including DEC [4] [11], Solbourne and Corollary.

However, the approaches taken by the implementors of these operating systems have varied widely. Some implementations have used a master/slave paradigm, others have used coarse-grained data structure locking, and a few have used finer-grained locking. One or two implementations rewrote Unix from scratch to accommodate multiprocessor support but most implementations have concentrated on adding coarse-grained parallelism to

existing sources. The Encore Mach Unix compatibility code evolved from the CMU Mach master/slave synchronization to coarse-grained and then to fine-grained locking protocols[2] [7] .

When OSF replaced the 4.3BSD compatibility code with 4.3BSD-Reno compatibility code, Encore and OSF were faced with the task of parallelizing the 4.3BSD-Reno Unix filesystem code (UFS). OSF/1 UFS re-uses much of the original Encore Mach UFS code but several aspects of the OSF/1 design and implementation are novel: the locking protocols and the single-stream performance optimizations. The OSF/1 UFS locking protocols differ materially from the original 4.3BSD-Reno protocols. We describe the Unix filesystem locking model in Section 2. Because we were also concerned with single-stream performance in OSF/1, we developed optimizations to eliminate the directory search and inode cache probe overhead inherited from Encore Mach. Sections 3 and 4 describe these optimizations and Section 6 shows their characteristic performance. In Section 5, we analyze races that resulted from the introduction of the OSF/1 locking model and its optimizations. We conclude in Section 7 with a summary of our experiences.

2. Unix File System

Source Code Genesis

The Unix File System in OSF/1 derives from the original Berkeley Fast File System introduced in 4.2BSD [8]. Sun placed the Fast File System code under their Virtual File System (VFS) [6]. VFS hides the details of various filesystem implementations beneath the vnode, an abstraction of a file. The rest of the kernel only knows about vnodes and calls through an array of function pointers known as the vnode switch to invoke filesystem-dependent functions. Thus, most of the kernel has no dependencies on any particular filesystem implementation.

Recently, 4.3BSD-Reno introduced its own VFS layer [5] and included a modified Fast File System as the local filesystem type.

All of the implementations mentioned above were targeted towards uniprocessor systems. Parallelized versions of the Fast File System began to appear in 1985 from a few sources [3] [10] [11] using a variety of locking protocols, with differing system targets and performance goals. Encore Mach first provided a parallelized version of the Fast File System based on blocking mutual exclusion locks [2], with minimal code modification. Preserving as much of the original code and data structures as possible allowed us to easily track changes in other source code bases, such as Berkeley and CMU. The resulting filesystem offered fairly good parallelism while not straying far from the original code base. However, significant bottlenecks remained in the system. For example, the use of a mutual exclusion lock prevented reads from executing in parallel against the same file.

Encore Mach subsequently shifted to a CMU Mach release that incorporated the Sun VFS-based filesystem. We exploited the opportunity to apply lessons from the Fast File System parallelization effort to the parallelization of the VFS and UFS layers [7]. We increased parallelism by replacing critical, blocking mutual exclusion locks with spin locks or read/write locks. Minimizing source code changes remained an important goal, however, to ease the problem of integrating updates from the various third-party source code bases.

The development of the OSF/1 UFS filesystem type relied heavily on work done in the most recent Encore Mach parallelization effort. We reused much of the knowledge gained from that experience as well as much of the code. However, the goals of the OSF/1 implementation differed from those of the Encore Mach implementations.

UFS Development Goals

First, OSF/1 UFS emphasizes single-stream, uniprocessor performance as much as it does good multiprocessor performance. Encore Mach versions traditionally emphasized the latter, sometimes at the expense of the former. We knew that OSF/1 would run on uniprocessor platforms ranging from personal computers to mainframes so we were greatly concerned with preserving (if not improving) the traditional, single-stream performance of UFS.

Second, we sought to implement a completely lock-based synchronization scheme for UFS in place of the original Unix synchronization model, which assumes only interrupt-related events can compete with process-context kernel activities for access to data structures. Interrupt-based synchronization works only on uniprocessors.

However, we also intended to build both uniprocessor and multiprocessor kernels from the same UFS code. We made every effort to avoid developing separate code for the uniprocessor and multiprocessor cases, to improve readability and simplify maintenance. In other words, only a few `#ifdefs` are uniprocessor- or multiprocessor-specific.

Maintaining common, lock-based sources for UFS did not imply that we preferred multiprocessor performance over uniprocessor performance. Maximizing performance for each case did not necessarily require separate code. On the one hand, uniprocessor performance improvements usually benefitted the multiprocessor case. On the other hand, the uniprocessor kernel has no need for locks used only for multiprocessor synchronization. We coded manipulations of these locks using macros, so that the lock manipulations disappear when building OSF/1 for a single-cpu target. Thus, the multiprocessor synchronization overhead may be eliminated for a uniprocessor platform.

Finally, we permitted ourselves somewhat greater latitude in modifying code and data structures in OSF/1 than in previous versions of Encore Mach. We still ruled out gratuitous modifications to the source base to simplify tracking OSF's source code donors. On the other hand, because we were building a new operating system we deemed clean uniprocessor and multiprocessor support more important than maintaining line-for-line compatibility with the original source code. For example, where absolutely necessary we carefully rewrote algorithms to support both uniprocessor and multiprocessor targets without requiring separate code for each.

Before we continue describing UFS, we must briefly mention a few properties of the OSF/1 virtual filesystem layer.

VFS Synopsis

The VFS layer places no restrictions on the synchronization primitives used by underlying filesystem implementations. For instance, one filesystem might rely on long-term, blocking mutual exclusion locks to control access to its files while another might use read/write locks or spin locks. However, because the VFS layer cannot predict what locks or other state might be accumulated by a filesystem, it does not permit filesystems to return to the VFS layer while retaining state information. Permitting the filesystem to hold onto such data would force VFS to guarantee callbacks to the filesystem in error cases. Retaining state, especially locks, that VFS does not understand increases the likelihood that VFS can cause the filesystem implementation to deadlock. Finally, minimizing the time during which locks may be held increases the parallelism in the system. For all these reasons, VFS imposes an essentially stateless model on its

underlying filesystem implementations.

However, a stateless filesystem model may not perform as well as a stateful model. Because the filesystem implementation cannot remember previous results, successive filesystem calls in a multiple call sequence may be forced to repeat error checks made in earlier calls. These checks may be expensive, particularly if they involve re-checking the contents of directories or on-disk state.

The OSF/1 VFS allows filesystem implementations to cache hints across multiple calls. However, the filesystem assumes responsibility for deciding whether the information returned to the VFS layer is still valid when the VFS layer passes it back to the filesystem on successive calls. Moreover, because the cached information is only a hint, the VFS layer has no obligation to return the data to the filesystem in the event of an error.

The filesystem can use these hints at its discretion to avoid redundant checking and thus reclaim single-stream performance that might otherwise be lost in a purely stateless implementation.

OSF/1 UFS Synchronization Model

UFS must conform to the restrictions placed on it by VFS, as detailed in the following paragraphs. Before returning to the VFS layer, UFS unlocks any locks and disposes of any other state it has accumulated. However, UFS also takes advantage of the VFS caching feature to record information about directory state during a pathname lookup and use these hints on succeeding calls back to UFS. Of course, a competing operation may alter the state of the directory between the lookup call and the following UFS call, forcing a re-examination of the directory state.

UFS employs two primitive lock constructs. UFS uses one such construct, the multiple-reader/single-writer locks, to synchronize access to the contents of a file or directory. The "I/O lock" resides in the in-core inode structure. When reading from the file, *ufs_read()* first acquires the inode's I/O lock for reading; when writing to the file, *ufs_write()* first acquires the inode's I/O lock for writing. This convention guarantees that all data in the file will be consistent when read and written but that multiple reads may proceed simultaneously on a single file. Because time-sharing systems tend to generate many reads, especially against common files and directories, using a read/write lock to guard file data instead of a mutual exclusion lock offers a substantial performance gain in the typical case [7]. Of course, UFS cannot hold the inode I/O lock when returning to the VFS layer. Unlocking the inode I/O lock allows new races on file and directory operations, which typically must be resolved by re-examining the state of the file or directory.

UFS uses spin locks, called simple locks in Mach, to guard most of its data structures. (When compiled for a uniprocessor target, simple locks disappear via the magic of macro substitution, thus maximizing performance.) These structures include the inode cache hash chains and freelist, in-core inodes, superblocks, etc. We hold these locks for brief periods of time; in fact, we rarely, if ever, hold them across a function call and we never hold them in situations where the kernel might block. As a result, simple locks may be released and re-acquired while operating on a single data structure. On a multiprocessor, new races may become possible on the data structures guarded by simple locks. Resolving such a race usually requires inspecting the data structure again after re-acquiring its lock.

Clearly, the OSF/1 UFS implementation must resolve several races that were not problems in the original, uniprocessor Unix filesystem. These races affect multiprocessor systems and also affect uniprocessor systems in cases where the kernel can block in between or during UFS calls. In the worst case, the duplicated checks to detect these races significantly degrade the performance of the affected operations.

UFS minimizes the need to duplicate checks by associating timestamps with affected data structures. A timestamp records the time at which a data structure is modified. By saving a structure's timestamp value and later comparing the saved value with the structure's current timestamp, we can quickly determine whether the structure has changed. Timestamps first appeared as a database performance optimization[13] but can easily be applied to the UFS problems described above. While timestamps commonly record actual time values, we implement timestamps as monotonically increasing 32-bit counters.

In particular, we applied timestamps to inode cache operations and directory operations in UFS. Our timestamps cannot eliminate the possibility of duplicate state checks but reduce the occurrence of those checks to acceptable levels as described in the next sections.

3. Inode Cache

The Unix File System maintains a cache of inodes to optimize inode lookups. The inode cache is composed of buckets, each containing a header with an attached list of inodes. Each inode hash chain header contains a spin lock that serializes access to that list and is held while the list is searched or manipulated. The hash chain locks are held for relatively short lengths of time, thus permitting a high degree of parallelism.

The *iget()* function looks up an inode in the inode cache. The basic algorithm for *iget* is:

1. Hash the inode into a hash bucket and acquire

the hash chain lock for the attached list.

2. Search the list for the inode.
3. If the inode is in the cache, release the hash chain lock and return the inode.
4. If the inode is not in the cache:
 - * Release the hash chain lock.
 - * Allocate a new inode.
 - * Re-acquire the hash chain lock, add the new inode to the chain, and release the lock.
 - * Read the on-disk portion of the inode and initialize the inode.
 - * Return the inode.

When an inode is not in the cache, *iget* allocates a new one. Inode allocation is a relatively time-consuming process that may require blocking, even on uniprocessors. Therefore, we cannot hold the hash chain lock while allocating inodes. Releasing the lock, however, opens a window where two threads can add duplicate inodes to the cache. Two threads may race to find the same inode in the cache in the following way. Each thread acquires the hash chain spin lock, in turn, and searches the chain before releasing the lock. Neither thread finds the inode in the cache, so both allocate new in-core inodes, re-acquire the hash chain spin lock, and attempt to add their new inodes to the cache. If *iget* does not detect this condition, the cache becomes corrupted with two copies of the same inode. Thus, using spin locks alone to guard the inode hash chains does not prevent duplicate inodes from entering the cache.

We can guarantee the uniqueness of inodes in the cache by rescanning the hash chain after allocating a new inode. In this scheme, *iget* re-acquires the hash chain lock after allocating an inode and searches the chain for a duplicate. If it finds one, it releases the new inode and hash chain lock and returns the inode from the cache instead. Otherwise, *iget* inserts the new inode into the cache before releasing the hash chain lock. This scheme is described in [7]. While this method ensures that the cache contains unique inodes, it also adds substantial overhead to the insertion algorithm.

In OSF/1, we wanted to avoid the additional overhead of rescanning the inode hash chain each time we allocate a new inode. In fact, we only need to re-examine an inode hash chain when one or more inodes are added to the list while a new inode is being allocated. We modified *iget* to use timestamps in the hash chain headers to detect additions to the chains as follows:

1. When the inode cache lookup fails, record the timestamp from the hash chain header before releasing the hash chain spin lock.
2. Allocate the new inode.
3. Re-acquire the hash chain spin lock and compare the current timestamp to saved value.
4. If the timestamp changed while the new inode

was being allocated:

1. Rescan the hash chain for a duplicate inode.
2. If a duplicate is found, release the hash chain lock, drop the new inode, and return the inode found in the cache.
5. Insert the inode at the head of the hash chain.
6. Increment the hash chain timestamp.
7. Release the hash chain spin lock and return the new inode.

We eliminated most inode hash chain rescans by using timestamps. We only check a hash chain's timestamp when inserting an inode into the cache, therefore we only need to increment that timestamp when adding an inode to that chain. As the statistics demonstrate in Section 6, timestamps infrequently change while inodes are being allocated. Thus, we rarely need to rescan the hash lists.

4. Directory Operations

Directory manipulations are a major part of the UFS layer. Ultimately, many pathname translations and all successful create, delete and rename operations in the Unix file system require directory accesses and modifications.

The OSF/1 VFS and UFS locking policy, as described in Section 2, has several implications for directories. In 4.3BSD-Reno, when an operation on a directory must be performed, the directory inode's mutual exclusion lock may be held across multiple UFS operations. By locking the directory inode, no operations can be performed on that directory by other threads between the UFS calls. Consider creating a previously non-existent file; the 4.3BSD-Reno algorithm is as follows:

1. During pathname translation, the VFS layer calls *ufs_lookup()* to convert a pathname component to a vnode.
2. *Ufs_lookup* returns to the VFS layer with a locked parent inode, as well as offsets into the parent directory where the new entry should be placed.
3. Eventually, the VFS layer calls *ufs_create()*, which then calls *maknode()*. *Maknode* allocates a new inode for the file and calls *direnter()* to add the directory entry into the parent directory.
4. *Direnter* uses the offsets from *ufs_lookup* to add the new directory entry and writes the directory to disk. *Direnter* then calls *iput()* to unlock the parent inode.

So, the lock on the parent directory inode was held from the pathname translation stage, through the leaf inode allocation and initialization, and until the directory was updated and written to disk.

The parallelized create algorithm, based on the Encore Mach algorithm, does not hold the parent directory inode locked for the entire operation. This scheme acquires the directory inode's read/write lock

only when we are using the directory and does not hold the lock outside the UFS layer. Therefore, during part of the create operation other threads are able to manipulate the directory.

1. During pathname translation, the VFS layer calls `ufs_lookup()` to convert a pathname component to a vnode.
2. `ufs_lookup` acquires the parent directory's inode read lock when scanning the directory. It determines the offset where the new entry should be placed. It releases the read lock on the parent and returns.
3. Later, the VFS layer calls `ufs_create` which then calls `maknode`. `Maknode` allocates a new inode for the file. It then acquires the write lock on the parent directory before calling `direnter`.
4. `Direnter` rescans the directory to get the current offsets. It then adds the directory entry and writes the directory to disk.
5. `Maknode` releases the write lock on the parent directory inode after `direnter` returns.

Notice the overall algorithm for entering a file into a directory remains largely unchanged. However, `direnter` must always rescan the directory to compute new offsets because the directory has been unlocked and may have been modified. Another thread could have added an entry at the offset saved by `ufs_lookup` or even added the same entry we are trying to add. Unfortunately, every `direnter` operation now becomes much more expensive.

As with the inode cache, described in Section 3, we use timestamps to reduce the number of directory rescans. These timestamps are located in directory inodes. We increment the directory inode's timestamp whenever we modify the contents of the directory. In `ufs_lookup`, we record the timestamp of the parent directory inode, while holding its read lock. Then, when we finally want to add the entry we compare the saved timestamp with the parent's current timestamp, while holding the parent's write lock. If the timestamp hasn't changed, no other thread has manipulated the directory, and so the offsets cached by `ufs_lookup` remain valid. If the timestamp has changed, we must rescan the directory to obtain current offsets. Here is the modified `direnter` algorithm:

1. Compare the timestamp cached by `ufs_lookup` with the parent's current timestamp.
2. If the timestamps do not match, rescan the directory to obtain the current offsets.
3. Add the directory entry and write the directory to disk.

Like create, destructive operations such as `rmdir` and `unlink` must check directory timestamps and possibly rescan. When deleting a file, it is possible that another thread is racing to delete the same file. (This race condition exists in every Unix system.) Both threads find the entry in `ufs_lookup`, and return

with the offsets for the entry to be deleted. However, only the first thread to execute `dirremove()`, the function that removes an entry from a directory, will succeed in removing the entry while holding the directory inode's write lock. The second thread calling `dirremove` discovers that the timestamp has changed and does not find the file when rescanning the directory. The second thread will return the appropriate error code as if the file had never existed.

The statistics in Section 6 demonstrate that applying timestamps to directory operations substantially reduces the number of directory rescans.

5. New Race Conditions

The OSF/1 locking model has introduced new races during directory operations. Some races were created by eliminating inode locking from the vnode layer; others were created by introducing directory timestamps. The UFS functions handling vnode operations (e.g. file creations and deletions) catch the first set of races, while the functions that perform directory operations detect the second set. Both types of races can occur on uniprocessors as well as on multiprocessors.

Vnode Operation Races

As mentioned in Section 2, the 4.3BSD-Reno kernel locks inodes indirectly from the VFS layer and they may remain locked for long periods of time. This model prohibits other threads from reading or updating a directory while its inode is locked, effectively serializing all directory operations. The OSF/1 kernel only acquires the inode read/write lock in the UFS layer; thus inodes are locked for a shorter time than in the 4.3BSD-Reno model. However, this more sophisticated locking protocol opens windows where several threads may perform operations on the same directory in parallel, giving rise to several new race conditions. The OSF/1 UFS layer detects these races using the following techniques:

- * Intelligent use of read/write locks on directory inodes.
- * Link count checks on directory inodes.
- * Directory rescans when the directory timestamp as changed between the pathname translation and the directory operation.

The following example describes a race condition created by the OSF/1 locking scheme and shows how we apply the first two techniques to detect the race and serialize directory modifications. We describe an application of the last technique in the Section titled "Directory Timestamp Races."

Many races in the UFS layer may occur when removing a directory. Suppose a thread, A, is removing directory `"/foo/bar"` while another thread, B, is attempting to create a file, `"/foo/bar/junk"`. We must prevent thread B from creating a file in the

directory if thread A is in the process of removing it and prohibit thread A from removing the directory if thread B is in the process of creating the file.

The basic algorithm for removing a directory is:

1. Call *namei()* to perform pathname translation.
2. Call *ufs_rmdir()* to remove the directory.
3. The functions in the UFS layer do the following:
 - * Acquire the inode write lock on the parent directory, followed by the inode read lock on the child directory.
 - * Verify that the child directory is empty, remove the directory entry from the parent directory, and increment the timestamp on the parent directory inode.
 - * Decrement the link count on the parent directory by 1 and release its inode write lock.
 - * Decrement the link count on the child directory to 0 and release its inode read lock.
 - * Truncate the inode.

The algorithm for creating a file is:

1. Allocate a file descriptor.
2. Call *namei* to perform pathname translation.
3. Call *ufs_create()* to create the file.
4. The functions in the UFS layer do the following:
 1. Allocate and initialize a new inode.
 2. Acquire the inode write lock on the parent directory.
 3. Write out the new inode.
 4. Create a new directory entry and write the directory to disk.
 5. Increment the timestamp on the parent directory inode and release the write lock.

Both threads may perform pathname translations in parallel, but they must then modify the directory in serial. Thread A may verify that *"/foo/bar"* is empty, remove the directory entry, and even truncate the directory after thread B has translated its pathname but before thread B attempts to create *"/foo/bar/junk"*. The UFS layer must detect that the directory has been removed and prohibit thread B from creating the file. Also, thread B may create *"/foo/bar/junk"* before thread A removes *"/foo/bar"*. In this case, the UFS layer must detect that the directory is no longer empty and prevent thread A from removing it. The UFS layer detects these conditions by using inode read/write locks and checking link counts on directory inodes.

When removing a directory, we hold the inode write lock on the parent to prevent simultaneous modifications to that directory. We also acquire the inode read lock on the child directory, forcing any threads modifying the directory to block. This

prevents thread B from creating *"/foo/bar/junk"* between the time that thread A determines that *"/foo/bar"* is empty and the time it removes the entry. However, thread B may look up *"/foo/bar"* before thread A removes it. Then between the time thread A removes the entry and the time it truncates the directory, thread B may call *direnter* to create an entry for "junk" using the in-memory copy of *"/foo/bar"*. However, *direnter* checks the link count on the directory and determines that it has been removed. These two techniques, inode read/write locks and link count checking, prevent thread B from creating *"/foo/bar/junk"* after thread A removes *"/foo/bar"*.

We also prevent thread A from removing *"/foo/bar"* after thread B has created *"/foo/bar/junk"*. We hold the write lock on the parent directory inode when creating files. If thread A attempts to read *"/foo/bar"* after thread B has acquired the inode write lock, it blocks until thread B has created "junk". It then acquires the inode read lock on *"/foo/bar"* and determines that the directory is not empty. Thus, only the first thread acquiring the inode read/write lock on the directory succeeds and no directory modifications are lost.

Directory Timestamp Races

The second type of race was created by the introduction of timestamps for directory inodes. We originally implemented the directory timestamp as two separate timestamps, one for additions and another for deletions. The rest of this section explains this implementation, some of the races it created, and why this optimization failed to work as expected.

The dual timestamp method works as follows:

- * When creating a file:
 1. Check the directory addition timestamp before creating the file.
 2. Rescan the directory if the addition timestamp has changed since the pathname translation.
 3. Increment the addition timestamp.
- * When removing a file:
 - * Check the directory deletion timestamp before deleting a file.
 - * Rescan the directory if the deletion timestamp has changed since the pathname translation.
 - * Increment the deletion timestamp.

This method seems desirable because it may produce fewer rescans than the single timestamp method by separating file creations and deletions. However, the subsequent examples describe basic flaws in this scheme. First we must digress and discuss directory structures.

A directory contains a variable number of entries, each of which includes a file name, a file name length, and a record length. The record length is the number of bytes between the start of the directory entry and the start of the next entry. A record may be much larger than the file name length because each directory entry may contain unused space. For instance, the last entry in a directory block contains all the free space until the end of the block.

We use the information contained in the directory entries when performing pathname translations. When a thread looks up a file for deletion, it acquires the inode read lock on the parent directory and records the following information:

- * The directory offset for this entry.
- * The byte offset from the previous entry in the directory block, if there is one.
- * The directory deletion timestamp.

Later, the VFS layer calls `ufs_remove()` to remove the file. The functions in the UFS layer do the following:

1. Acquire the write lock on the parent directory inode.
2. Examine the directory deletion timestamp.
3. If the deletion timestamp has changed since the pathname translation, rescan the directory and record the current values of:
 - * The directory offset for this entry.
 - * The byte offset from the previous entry in the directory block, if one exists.
4. Remove the directory entry and collapse the new free space into the previous entry in the directory block, if there is one.
5. Release the write lock on the parent directory

inode.

The following example illustrates why it is insufficient to check the deletion timestamp when removing a file. Suppose a directory contains entries for `“.”`, `“..”`, `“foo”`, `“bar”`, and `“junk”`, as shown in Figure 1. (The figures in this section represent free space in directory entries by cross-hatched regions.) A thread removing `“bar”` calls `ufs_lookup` to determine the offset of that entry in the directory. `Ufs_lookup` also records the deletion timestamp and the distance between `“bar”` and the previous directory entry, `“foo”`. Later, the UFS layer checks the deletion timestamp and rescans the directory, if necessary, to record the current byte offset from `“bar”` to the previous entry. Then `ufs_remove` deletes the directory entry for `“bar”`, locates the previous directory entry using the distance saved during the pathname translation or rescan, and adds the new free space to the directory entry for `“foo”`.

Now suppose that another thread creates a file `“stuff”` between the time the deletion timestamp is recorded and the time it is checked, as shown in Figure 2, and no files have been removed from the directory during this time. When the thread removing `“bar”` checks the deletion timestamp, it has not changed, so the directory is not rescanned. But the distance from the previous directory entry recorded during the pathname translation is no longer valid. That distance was calculated when the entry directly before `“bar”` was `“foo”`, not `“stuff”`. `Ufs_remove` uses that saved distance and adds the new free space to the end of the directory entry for `“foo”`. Thus, the directory entry for `“stuff”` is lost, as illustrated in Figure 3. Therefore, when removing a file, we

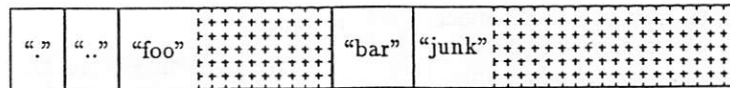


Figure 1: Original Directory

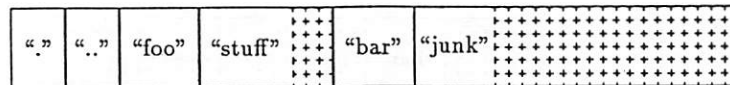


Figure 2: Directory After Creating "stuff"

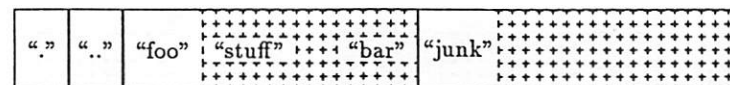


Figure 3: Directory After Removing "bar"

must check both the directory addition and deletion timestamps and rescan the directory if either of them has changed since the pathname translation.

The next example shows why it is necessary to check both the addition and deletion timestamps when creating a file.

When looking up a file for creation, threads call `ufs_lookup()` to perform the following:

1. Search the directory for an entry with enough free space to accommodate the new entry.
2. Record the byte offset of the located entry, if one exists.
3. Record the directory addition timestamp.

Later, the VFS layer calls `ufs_create()` to create the file. The functions in the UFS layer do the following:

1. Acquire the write lock on the parent directory inode.
2. Examine the addition timestamp.
3. If the addition timestamp has changed since the pathname translation, rescan the directory and record the byte offset of a directory entry with enough free space to contain the new entry, if one exists.
4. If a directory entry with enough free space was located, trim the large entry and create the new entry.
5. If no directory entry was large enough, allocate a new directory block and create the new entry.
6. Release the write lock on the parent directory inode.

A thread attempting to create a file "stuff" calls `ufs_lookup` to obtain the directory offset where the file can be created and to record the addition timestamp. Suppose `ufs_lookup` determined that the directory entry for "junk", the last one in the block, had enough unused space to contain the directory entry for "stuff". Now suppose another thread removed the file "junk" between the time the addition timestamp was recorded and the time it was checked, as shown in Figure 4, and no other files

were created during this time. The addition timestamp has not changed, so the directory is not rescanned and the UFS layer splits the deleted entry for "junk" to create the new entry for "stuff". But "bar" is now the last entry in the block and it contains all the unused space until the end of the block. So the new entry created for "stuff" gets lost, as shown in Figure 5. Therefore, when creating a file, we must check both the directory addition and deletion timestamps and rescan the directory if either of them has changed since the pathname translation.

Since we need to check both timestamps on directory additions and deletions, there is no use for separate timestamps. Thus, we collapsed the two timestamps into a single timestamp that is always saved during pathname translations and later examined by the UFS layer. If the timestamp has changed since the pathname translation, the UFS layer rescans the directory. The single directory timestamp method eliminates the directory operation races introduced by using two timestamps without adding much overhead. The statistics presented in Section 6 confirm that we rarely rescan directories when using one directory timestamp per inode.

6. Analysis of Timestamp Optimizations

Overview

After adding our directory and inode cache timestamp optimizations, we sought to determine their effectiveness. In particular, we were interested in the number of rescans that had to be performed, that is, the number of times we were forced to re-examine a directory or rescan an inode hash chain. We settled on a standard hardware and software configuration and a simple benchmark.

Test Environment

Hardware

We used an Encore Multimax with the following hardware resources for all of our tests:

- * Memory -- 72 megabytes
- * Processors -- 6 National Semiconductor 32532

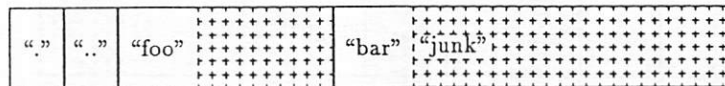


Figure 4: Directory After Removing "junk"

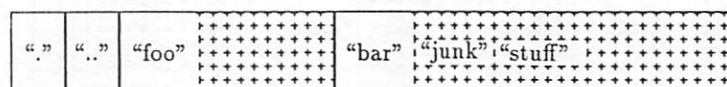


Figure 5: Directory After Creating "stuff"

processors at 25 Mhz (7.5 VAX MIPS each, 45 MIPS total)

- * Disk Interface -- 1 Ethernet/Mass Storage Card, providing one SCSI channel at 1.5 Mbytes/sec

- * Disk -- one 624MB NEC disk

The Multimax is a tightly-coupled, globally shared-memory multiprocessor.

Software

The operating system software was OSF/1 build 25, approximately equivalent to the version of OSF/1 Release 1.0 shipped in December, 1990. We modified our version to include the performance instrumentation described below but all of the filesystem functionality, including the timestamp optimizations, was already a standard part of OSF/1.

Test

The test consisted of continuously running 6-way parallel kernel builds for several hours from single-user mode (with network disabled) with the machine configured for six processors. We later reran the test with the machine configured as a uniprocessor. The kernel running this test supported 512 inode hash chains and 2204 vnodes. The kernel builds were executed in a continuous loop of 'make clean' followed by 'make'. These kernel builds use a version of make modified to process its dependency graph in parallel where possible. In short, the parallel make knows how to execute multiple source file compilations in parallel by simultaneously invoking several copies of the compiler. The compiler itself is an ordinary single-stream C compiler. The tests built Encore Mach kernels, where each compilation phase actually requires three programs connected by pipes, as follows:

```
cc -S kernel_file.c | \
  inline | as -o kernel_file.o
```

On a multiprocessor, OSF/1 will run all three programs on separate processors, simultaneously, if each program is ready to run and processors are available. Thus, running an N-way kernel build actually yields up to N*3 active processes.

A parallel kernel build is an effective test of our optimizations because it creates the opportunity for many collisions on common directories: source directories, the build directory where the results of the compilations are stored, and */tmp*.

Inode Cache Timestamps

A global inode cache statistics structure contains summaries of operations on the inode cache.

```
struct icache_stats {
    /* iget started over from scratch */
    u_int ic_iget_loop;
    /* cache hits */
    u_int ic_iget_hit;
    /* iget rescanned hash chain */
    u_int ic_iget_rescan;
    /* insertions into cache */
    u_int ic_iget_insert;
}
```

The *ic_iget_loop* variable records the number of times *iget* is called or restarted; in essence, the number of probes into the cache. The *ic_iget_hit* variable counts the number of times *iget*'s probes are successful. *ic_iget_rescan* counts the times *iget* rescans the inode hash chains looking for duplicate inodes. *ic_iget_insert* records the number of times an inode was allocated and inserted into the cache. Note that the number of insertions added to the number of hits does not necessarily yield the total number of probes into the cache because our disk checking method at boot time uses *iget* in a special way that probes the cache without inserting any inodes.

After running the build repeatedly for five hours on a six processor configuration, the kernel accumulated the following inode cache statistics:

```
Total probes (ic_iget_loop): 174978
Number of hits (ic_iget_hit): 131621
Number of rescans
  (ic_iget_rescan): 0
Number of insertions
  (ic_iget_insert): 43349
```

We also ran the test in a uniprocessor configuration, yielding the following results:

```
Total probes (ic_iget_loop): 34341
Number of hits (ic_iget_hit): 25370
Number of rescans
  (ic_iget_rescan): 0
Number of insertions
  (ic_iget_insert): 8964
```

About 25% of all cache probes in both cases caused a new inode to be allocated and inserted into the cache, yet none of the chains had to be rescanned. However, we cannot eliminate the rescan check because the inode insertion race we described earlier does exist; but the timestamp optimization eliminates almost all of the burden of redundant inode hash chain scans.

Directory Statistics

We implemented statistics gathering for all directory manipulations to determine what effect, if any, timestamps have on directory searches. A global *dir_stats* data structure maintains the following counts:

```

struct dir_stats {
    /* total directory operations */
    u_int dir_ops;
    /* needed to rescan directory */
    u_int dir_rescan;
    /* insert operation already happened */
    u_int dir_exist;
    /* remove operation already happened */
    u_int dir_rm;
    /* max rescans per inode */
    u_int max_rescan;
    /* inode dev that has max rescans */
    u_int max_dev;
    /* inode num that has max rescans */
    u_int max_inum;
};

```

The *dir_ops* field records the total number of directory modification attempts that have occurred. The *dir_rescan* field contains the number of times a directory rescan happened because the timestamp changed between the lookup operation and the actual create or remove operation. The *dir_exist* and *dir_rm* records indicate the number of rescans that occurred because the desired operation had already taken place, i.e. two or more threads were racing to create or remove the same directory entries. We maintain the device and inode number of the directory with the most rescans in *max_dev* and *max_inum* respectively. This information aids us in quickly identifying that directory. We also record its rescan count in *max_rescan*.

One of the primary pieces of information we wanted to determine was the effectiveness of the directory timestamp. Figure "Directory Statistics" shows the number of rescans as a percentage of the total number of directory operations. We plotted the first four and a half hours of a continuous kernel build. We expect that the number of directory operations and the number of rescans for every build and clean cycle will remain fairly constant. This

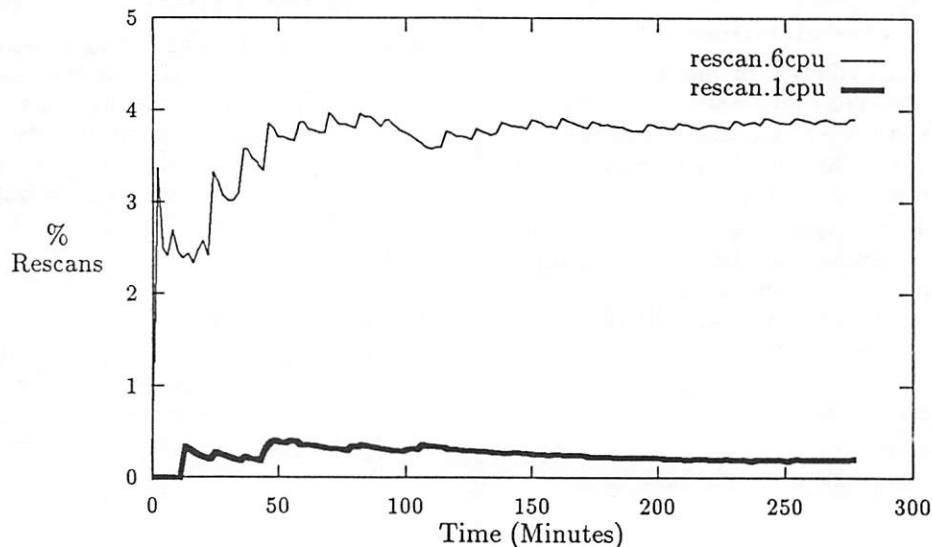


Figure 6: Percentage of All Directory Rescans

expectation arises from the repetitive and unchanging nature of the test. Therefore, we expect that the rescan percent will also become a constant. After several hours of running the rescan rate remains fairly constant between 3.8% and 3.9% for six processors. The rate for the uniprocessor remains close to zero, hovering near 0.2%.

The several sharp increases seen with six processors, early in the test cycle, occur when a 'make clean' happens. Since the make program is parallelized, the clean is also parallelized. Therefore, many threads race to delete the same object files. The snapshots of the directory statistics taken once every second during a clean, presented in Figure 7, show that every rescan results in discovering that the entry has already been deleted. However, the effects of these rescans become negligible in the overall pattern after several hours of testing.

Ops	Rescan	Exist	Rmvd	MaxRscn
0	0	0	0	0
10	2	0	2	0
13	4	0	4	0
8	2	0	2	0
12	3	0	3	0
15	5	0	5	0
13	4	0	4	0

Figure 7: Rescans during a 'Make Clean'

Every test showed that */tmp* was clearly the directory that had the most rescans. We expected this to be the case, since every compilation uses temporary files and the chances for collisions seem quite high. We suspect that the directory with the second highest number of rescans is the kernel build

directory. Although we do not save the second highest inode identification we suspect the build directory because of the behavior exhibited during cleans. Also, the chance seems reasonably high that multiple threads are trying to create different object files in the build directory which accounts for the additional rescans on an otherwise "idle" system.

Figure 8 supports the idea that the number of rescans on /tmp, as a percentage of total directory operations, moves towards a constant. Extending the data out several hours shows that the rate continues to remain constant. Comparing the uniprocessor results in Figure "Directory Statistics" to Figure "Directory Statistics" shows that all rescans were, in fact, rescans on /tmp. Also, the six processor results in the same figures show that the overall rescan percentage jumps fairly high, around the 25 minute mark, while the rescan percentage on /tmp drops during that same period. This data also supports the fact that all of the rescans during a clean occur in the kernel build directory and not in /tmp.

The results in the graphs indicate that over 96% of all directory operations did not require rescans.

7. Summary

We have demonstrated that the use of timestamps has optimized the directory search and inode cache probe operations in both the uniprocessor and multiprocessor OSF/1 implementations. Additionally, timestamps greatly reduce or eliminate the performance penalty associated with the new UFS locking protocol. Our statistics reveal that the rescan rates were very low even on directories experiencing high contention. Therefore, using timestamps is an excellent mechanism for avoiding the overhead of unconditional rescans.

References

- [1] M. Bach and S. Buroff. Multiprocessor UNIX Operating Systems. *AT&T Bell Laboratories Technical Journal*, 63:1733-1749, October 1984.
- [2] J. Boykin and A. Langerman. The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis. In *Workshop Proceedings, USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 105-126, El Toro, CA, 1989. USENIX Association.
- [3] Encore Computer Corporation, Marlborough, MA 01752-3089. *UMAX 4.2 Programmer's Reference Manual*.
- [4] G. Hamilton and D. Code. An Experimental Symmetric Multiprocessor Ultrix Kernel. In *Conference Proceedings, 1988 Winter USENIX Technical Conference*, Berkeley, CA, 1988. USENIX Association.
- [5] M. J. Karels and M. K. McKusick. Toward a Compatible Filesystem Interface. In *Conference Proceedings, Autumn 1986 EUUG Conference*, Buntingford, Hertfordshire, UK, 1986. EUUG.
- [6] S. R. Kleinman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Conference Proceedings, 1986 Summer USENIX Technical Conference*, pages 238-247, El Toro, CA, 1986. USENIX Association.
- [7] A. Langerman, J. Boykin, S. LoVerso, and S. Mangalat. A Highly-Parallelized Mach-based Vnode Filesystem. In *Conference Proceedings, 1990 Winter USENIX Technical Conference*, pages 297-312, El Toro, CA, 1990. USENIX Association.
- [8] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*,

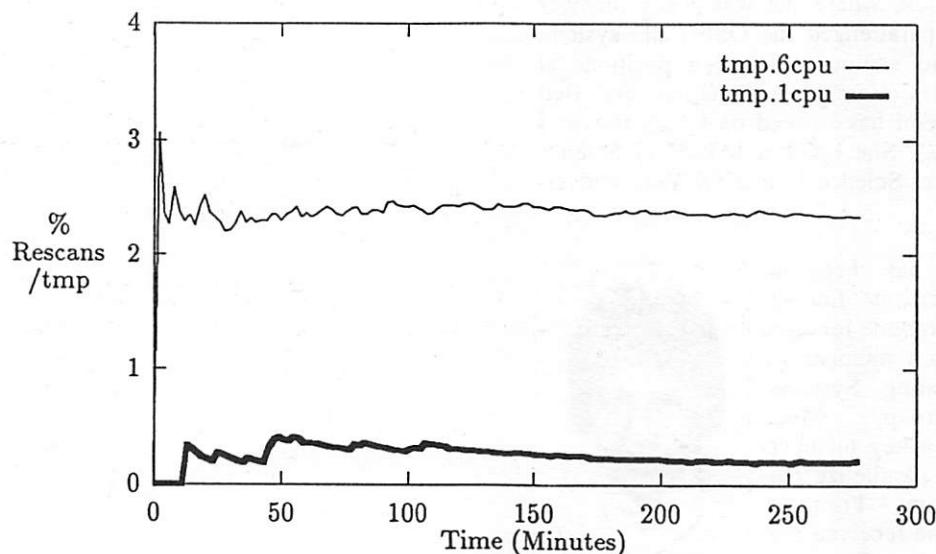


Figure 8: Rescans on /tmp

- 2:181-197, August 1984.
- [9] R. Rashid. Threads of a New System. *Unix Review*, August 1986.
 - [10] Sequent Inc., Beaverton Oregon. *Balance Guide to Parallel Programming*, 1986.
 - [11] U. Sinkewicz. A Strategy for SMP ULTRIX. In *Conference Proceedings, 1988 Summer USENIX Technical Conference*, pages 203-212, El Toro, CA, 1988. USENIX Association.
 - [12] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. Mach Threads and the Unix Kernel: The Battle for Control. In *Conference Proceedings, 1987 Summer USENIX Technical Conference*, pages 185-198, Berkeley, CA, 1987. USENIX Association.
 - [13] J. D. Ullman. *Principles of Database Systems (Second Edition)*. Computer Science Press, 1982.

Alan Langerman leads the OSF project at Encore Computer Corporation, where he has toyed with OSF/1, Mach and Unix dialects for the past several years.

George Feinberg received his MS in Computer Science from New Mexico State University in 1984. During a stint with Hewlett-Packard in Fort Collins, Colorado, George worked on networking and distributed file systems. He then moved to the Open Software Foundation in Cambridge, Massachusetts, where he has been involved with OSF/1 development in the areas of file systems and networking. George was a key member of the team that parallelized the OSF/1 file system.

Noemi Paciorek is a principal software engineer in the Mach Operating System Group at Encore Computer Corporation. Prior to joining Encore, Noemi was a principal software engineer at The Open Software Foundation, where she was a key member of the team that parallelized the OSF/1 file system. She has also held senior engineering positions at Computer Consoles, General Automation, and Bell Laboratories. Noemi has worked on UNIX internals for over six years. She holds a Master of Science degree in Computer Science from New York University.

Susan LoVerso has been a software engineer at Encore Computer Corporation for the past 3.5 years, as a member of the Mach Operating Systems development group. Most recently she was a key member on the project to parallelize the OSF/1 file system. Prior to joining Encore, she received her Masters degree in Computer Science from the State University of New York at Buffalo.



Advancing Files to Attributed Software Objects

Andreas Lampen – Technische Universität Berlin

ABSTRACT

The system described in this paper is part of the kernel of a UNIX based generic Software Engineering Environment (SEE) that shall be open to extension and integration of new as well as existing tools. Basis for integration is the notion of *software object* in the sense of identifiable and controllable piece of information evolving during software development. AtFS¹ (*Attributed File System*) is an *extension* to the UNIX file system to make it suitable as a repository for software objects.

The story of AtFS began as part of the shape-toolkit, a collection of programs supporting version control and software configuration management. AtFS's key issues were the ability to store multiple versions of files, a mechanism for attaching any number of application defined attributes to each version, and nonunique identification of versions by any attributes. It provides a consistent view of *Attributed Software Objects* (ASOs) that can either be immutable saved versions or regular UNIX files.

When adapting AtFS to be part of a generic SEE, additional requirements cause conceptual extensions. The notion of attributed software object is extended, beside *file versions*, to *histories* and *directories*. A concept for object identity, coming along with *persistent unique identifiers* for ASOs is introduced. The attribution mechanism is extended to typed and structured application defined attributes. The conceptual extensions lead to a totally new implementation of AtFS, supporting network distributed management of ASOs and featuring a revised interface. The interface new is designed in an object oriented manner.

Introduction

The notion of *file* has a long tradition in data management systems. It is often the basis for the organization of information and the smallest controllable and exchangeable piece of information in data management systems. A file contains a bundle of coherent contents data and has some control information tagged on.

Computer *file systems* take pattern from the traditional look of paper files, mostly providing additional (hierarchical) structuring facilities. Though sticking too tight to the tradition of paper files, most computer file systems support only part of the range of functionality electronic storage of files would make possible. Significant advantages of electronic files in respect to paper files are

- fast retrieval by multiple identification criteria,
- efficient management of multiple versions (good support for finding differences between versions) and
- easy transport and easy replication with potentially world wide sharing of files.

Especially the first two points lack support by most existing computer file systems. While even paper files allow (although not *support*) storage of multiple versions of their contents, only few file systems provide version control functionality. More seriously, the amount of control information stored together with a file is mostly limited to a certain number of fields with fixed semantics. No extensions possible.

Control information for files is denoted as *file attributes* or *file properties*. Typical file attributes in file systems are "owner", "size" or "date of last modification". File system applications often would be happy about the possibility to associate additional, application specific attributes with files. Just to name a few examples. A version control system introduces version numbers and version states. A bulletin board system keeps origin, topic and expiration date with each file. A file containing an electronic letter has a sender and a recipient. An object code file should have the generating compiler and the used compilation flags tagged on. The list is sheer endless.

Beside *administrative* information as above, attributes are also the appropriate means to store *descriptive* information about files. These are for example keywords related to the file's contents or a classification of the file according to a given taxonomy. Descriptive attributes, as well as administrative attributes, form the basis for a much more

¹Formerly, we used the acronym "AFS" which often lead to confusion with CMU's Andrew File System, so we changed the name slightly.

sophisticated retrieval functionality than most existing file systems provide. Attributes enable nonunique identification of files by giving attribute patterns while file systems typically require identification by location (path name).

In this paper we will present an approach to improve existing file systems. The main improvements concern the storage of any amount of control information with files, the management of multiple versions of files, and a concept for file identity, necessary for managing network distributed filing structures. The approach may partly be characterized as introducing techniques known from databases to file systems. This is true as we will provide a basis for realizing relationships between files and for implementing sophisticated retrieval techniques. Different to the database approach, the file system idea of strictly separating contents data and control information is retained. And – even more important – there is no need for a data schema. The issue of comparing database technology and file systems is discussed in more detail in the following section about data management in software engineering environments.

Key issue in this paper is the extension of the notion of *file* to *Attributed Software Object (ASO)*. An attributed software object is a file system object with a – potentially unlimited – number of attributes attached to it. In the presented model, an ASO may be

- a version of a regular file (*version*),
- a collection of file versions (*history*), or
- a collection of histories and directories (*directory*).

Versions take the place of files containing application specific contents data. *Histories* realize version control facilities. A history contains a set of related versions representing different states of one *conceptual* file. *Directories* serve for tree structuring as in conventional file systems.

Later in this paper we present the *Attributed File System (AtFS)*, an implementation of the ideas described above. AtFS is an extension to the UNIX file system that adds features like management of multiple versions and storage of any number of attributes to UNIX files. Additional major enhancements include a concept for location independent object identity (the basis for realizing relationships), network distributed histories, and a network wide protection mechanism.

Management of Data in Software Engineering Environments

This section gives a look at the background of the work described in this paper. AtFS is part of the kernel of a UNIX based generic software engineering environment (SEE). The kernel shall be basis for

the integration of new as well as existing tools supporting all kinds of needs in a software development project. Different techniques for integration are explored. These include tight integration of groups of dedicated tools developed for a common fine grain data design, obtaining integrated appearance of tools by constructing a coherent graphical user interface, and “gluing” together existing tools by putting some kind of mortar in between.

AtFS’s function is to support the latter integration technique, we name it *integration by adaptation*, where existing UNIX tools are fit into an object oriented framework. For this, the environment kernel contains a system for defining a type hierarchy for *typed software objects*. This includes a language for object oriented description of software object types and their properties, feeding a *type directory*. Typed objects are mapped to attributed software objects in AtFS using the attribution mechanism. The maintenance of the ASO attributes according to the type directory is performed by an object oriented command interpreter. In the conclusion of this paper, we will have a closer look at this specific application of AtFS.

UNIX

UNIX is often considered to be the best software engineering environment (SEE) available. This is mainly because it provides a huge variety of existing software development tools, and a stable basis for the development of new tools. However, UNIX as a toolbox system lacks integration and homogeneity.

One reason for the lack of integration are drawbacks in the data storage capabilities. UNIX provides no reliable mechanism to store information *about* files additionally to the standard file attributes maintained by its file system. This problem has been realized by some people yet. The presumably most extensive approach to the problem of systematically storing information about files is described in [Mogul1986a].

Furthermore, the UNIX file system lacks systematic support for storing multiple versions of files. Version control systems have to be introduced as auxiliary tools, in most cases poorly integrated with related tools.

Another problem is file identification. Files are usually identified by their name, a mechanism that is unsuitable for modeling relationships between files. The identification key changes, when the file is moved. Even solutions working with file identification by the triple “host, file system, inode number” fail when a file is moved from one file system to another. Additional file identification problems arise with network distributed applications. Using for example NFS, it is sometimes not easy (at least quite time consuming) to find out, whether a

file `coma:/u/andy/foo.c` is the same as `blurp:/rusers/andy/foo.c`.

Nevertheless, why is UNIX so successful? The generality and continuity of its data organization scheme – files and directories – is one main reason for its success. Of course, files and directories as a common basis for structuring data is not very much, but at least there is a stable basis, a thing that most databases do not provide.

Databases

Many attempts have been made to use databases as data repository in SEEs and programming environments. Beginning with highly specialized stores for fine grain structured data such as in the GANDALF [Haberman1986a] environment to dedicated software engineering databases like Damokles [Dittrich1986a].

The main problem with databases is – and will always be – the schema. Tools working on data stored in a database have to be designed with respect to a given schema. Furthermore in most cases these tools have to be modified, or at least recompiled, when the schema changes. Usage of a database is only feasible for well explored work processes with a more or less fixed schema.

An advance in respect to robustness against schema changes has been made with the introduction of databases with object oriented schema definition language, such as GemStone [Bretl1989a]. For a general discussion of schema evolution in object oriented databases see also [Skarra1987a] and [Banerjee1987a]. Even the best mechanisms for schema evolution in object oriented databases have their limits. They also fail with major conceptual changes of the original data design.

Despite a lot of research in the area of programming and software engineering environments in the past, the software development process still bears a lot of imponderabilities. Hence the current trend in SEE construction realizes adaptability and extensibility as major design goals. Prominent examples are the *Arcadia* project, the *Field* environment, and the *HP SoftBench* system. These systems usually rely on file systems or object bases rather than databases.

Object Bases

What is an object base? The term *object base* is often equated with *object-oriented database* which itself is a fuzzy defined term. We divide between (1) databases that are suitable as data store for object oriented applications, (2) databases with an object oriented schema definition language (object oriented databases) and (3) object bases.

Object bases evolve from “a synthesis of ideas from file systems and databases” [Nestor1986a]. Object bases are repositories for *software objects* [Tichy1988a]. In software engineering terms, a software object is an identifiable, controllable piece of information. Software object is comparable to the notion of *file*. Hence, object bases deal with relatively coarse grain data structuring.

In the past, different approaches have been made to introduce object bases to software engineering environments. The *ODIN* [Clemm1986a] system is a production system for driving complex tool processes in a UNIX environment. It maintains a specialized, adaptable store of information about the files, the production machinery deals with.

PGRAPHITE [Wileden1988a] is a store for persistent typed objects. Objects types have to be described as graphs. From the graph description, an Ada implementation for the storage of objects of a specified object type can be generated. *PGRAPHITE* is part of the *Arcadia* project, an American joint project aiming at the development of advanced software environment technology.

The Portable Common Tool Environment (PCTE) [PCTE1988a] was developed in an Esprit (European) project as basis for the construction of integrated project support environments. PCTE contains an object management system featuring a file like notion of typed objects and type specific attributes for objects.

AtFS, the system described in this paper, clearly falls into the category of object bases although it represents a quite low level of abstraction. Differing from the systems mentioned above, AtFS does not support user defined types of objects and type specific attributes. It has, similar to the UNIX file system, a fixed schema.

As pointed out earlier, there will be an application of AtFS that is capable to realize a type hierarchy for software objects stored in AtFS. This provides, down to a certain level of object granularity, the functionality of an object oriented database. The advantage of this approach is that the underlying schema of attributed software objects remains fixed, independent from any schema changes in the system above. So the basic structuring of the data is always preserved, even if all schema information gets lost.

The Attributed File System

The Attributed File System (AtFS) is an implementation of the idea of advancing *files* to *attributed software objects*. It is an extension to the UNIX file system. One of the most important design criteria of AtFS is, that its applications shall be able to live peacefully together with conventional tools working on the UNIX file system. It shall not be necessary to modify or even relink any UNIX tool.

AtFS applications shall be able to access regular UNIX files as attributed software objects, while UNIX tools use the file system interface. This requirement implies, that UNIX files last as they are and that any access to regular files via the file system interface should not make AtFS data inconsistent.

Additional design criteria are robustness, security, and support for network distributed filing structures. Robustness and security are goals that would be much easier to achieve if AtFS could rely on dedicated kernel support. However, with view to easy installation and porting, we decided that AtFS should not require modifications to the operating system kernel.

A Bit of History

Our first ideas for constructing an attributed file system came up in 1987. Our task was it, to construct a common basis for for the shape-toolkit [Mahler1988a], a collection of programs supporting version control and configuration management. The shape version control system offers a functionality comparable to systems like RCS and SCCS with a more friendly user interface. The shape configuration management program basically offers the *make* functionality with significant enhancements allowing the management of multiple versions and control of variants of a system. Version management is supported by providing a procedure of identifying appropriate component versions that together form a meaningful system configuration by user supplied configuration selection rules. These rules have the form of an attribute pattern that has to be matched by versions to be selected.

To make the enhancements possible, the shape configuration management program has to have full access to all versions of the components and should be able to store control information with these versions. What we needed was a common basis for the version control system and the configuration management program that provides basic version control facilities and the possibility to store control information with the versions. Furthermore it should be possible, to identify versions by any attributes.

The first version of AtFS, described in [Lampen1988a], was designed to overcome the limitations of the UNIX file system that make it unsuitable for the job described above. AtFS provides a consistent view of attributed software objects that can either be saved versions or regular UNIX files. UNIX files that were created, modified, or translated by regular UNIX tools are accessible by the shape-toolkit through the AtFS interface. So we produced overlapping domains. On one hand the UNIX domain with files modified by UNIX tools and on the other hand the AtFS domain that comprises the UNIX files and the saved versions to be accessed in a

uniform manner by AtFS applications like the shape version control system and the shape configuration management program.

AtFS also unifies access to Attributes. Attributes comprise standard attributes as defined in the UNIX file system (protection, owner, modification date etc.), version control specific attributes (version number, version state) and application defined attributes having the general form *name=string*. Each Software object in AtFS carries standard and application defined attributes. This especially implies, that even regular UNIX files can have any number of application defined attributes tagged on.

The second generation

Although AtFS was primarily invented to serve the shape-toolkit as a basis for storing and exchanging information, we very early realized, that it could be basis for a much broader spectrum of tools. In especially the combination of ordinary UNIX tools together with tools working on versions and attributes seemed to be challenging.

Our focus of research changed from pure configuration management support to mechanisms for constructing coherent software engineering environments by integrating new and existing tools. AtFS was revised and evolved from a dedicated basis for a system supporting software configuration management to a general data management system to be part of the kernel of a generic software engineering environment.

Second generation AtFS takes over the concepts from the first version and introduces additional features to complete the functionality.

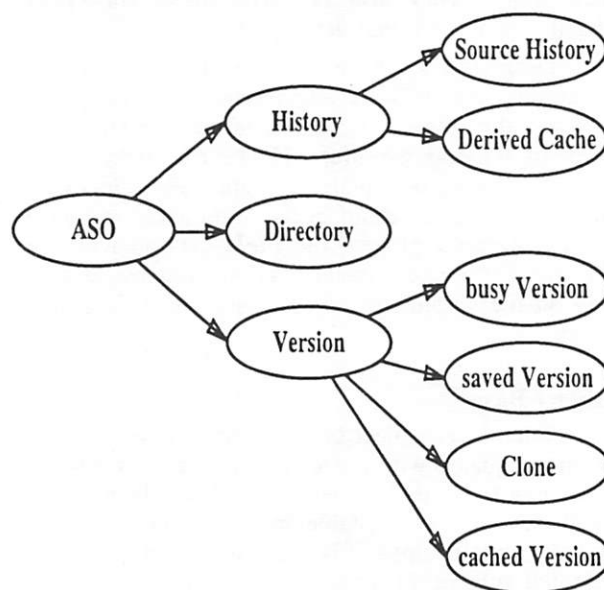


Figure 1: ASO types

Attributed Software Objects

Originally we tried to keep the data model of AtFS as simple as possible. Attributed software objects were always file versions with file like contents. Histories were not treated as autonomous attributed software objects but rather as some kind of special concept. This had the disadvantage, that no history attributes (version independent attributes) could be stored. The same with directories. The directory concept was taken from the UNIX file system with no possibility to tag attributes to directories.

In the revised data model as presented in this paper, the term attributed software object has a more general meaning. ASOs can be directories, histories, or single versions. An important issue added to the concept of ASOs is a concept of object identity. An ASO, once created has an location independent, persistent, unique identifier. Figure 1 shows a part of the data model of AtFS – the part dealing with ASOs – in form of a type hierarchy. The arrow in reverse direction can be read as *is-a*.

Unique Identifiers

In first generation AtFS, similar to the UNIX file system, ASO identification is location dependent. ASOs are uniquely identified by path name plus version number. When an object is moved, it's identifier changes. By this, this mechanism is especially insufficient for modeling references.

Furthermore, in first generation AtFS, the pair path name/version number needs not necessarily to be unique. Derived caches, these will be discussed in more detail later in the paper, might contain multiple versions with the same version number. In this case, as first generation AtFS is concerned, the distinction of versions has to happen by any attribute. It might even happen that two versions in a cache are identical by their standard attributes and can only be distinguished by application defined attributes.

We need unique identifiers (UIDs) for ASOs that are *location* independent and *representation* independent (independent of the attributes) [Khoshafian1986a]. As AtFS is redesigned also to be used in local and wide area networks, UIDs shall additionally be worldwide unique.

Second generation AtFS features 96 bit unique identifiers for ASOs. These are constructed from the host ID (the internet number of the host) the object was created on, the process ID of the creating process, the date of creation (seconds since 1970) and a serial number that is maintained by each process in order to ensure that the unique identifier is really unique even if two or more objects are created within one second. The unique identifier of an ASO will never be changed, even if the ASO is moved to another host. Each directory and each history has an own unique identifier. Versions inherit the unique

identifier from the history they belong to and additionally get an history-unique version number. The version number is unique also for versions stored in a derived cache.

Unique IDs are generated either on creation of a directory/history by means of AtFS or when first accessing a directory/history that was formerly created as a UNIX file system object. Creation of a history by means of the file system happens by simply creating a regular file. This automatically opens a new history if the name is new in the directory.

ASO Location

The unique identifier as described above does not contain any location information beside the creating host, which is potentially useless. Access to an ASO requires an *ASO handle*, a thing that consists of an unique identifier and a *location hint*. The location "hint" has it's name because AtFS does not provide any referential integrity for ASO handles. That means, after an ASO was moved, all ASO handles for this objects point to a wrong location. AtFS in this case, rather than trying to update all ASO handles which would be a nearly impossible task, leaves a pointer to the new location behind the moved ASO. So, when accessing or trying to access an ASO that has been moved in the meantime by use of an ASO handle, AtFS is able to update the location hint by following the track. ASOs that are moved by means of the UNIX file system, without leaving a track behind, are lost.

The location hint is built from the server the object resides on, the file system on the server, where the object physically lies and the relative path name on the file system. This mechanism makes AtFS immune against reconfiguration/rearranging of file systems on a server. In the case that a disk gets disconnected from one server and hooked up to another, AtFS can easily provide a mapping of location hints from the old server to the new one.

Histories

A history is a collection of versions. Generally, AtFS distinguishes between histories of source versions and derived versions. Source versions are objects that are created manually by a human. A history of source versions is organized as a tree representing the physical evolution of the versions. Derived versions are created automatically by a tool and may be recreated at any time. Histories of derived versions are organized simply as sets of related versions without any evolution structure. They are maintained as a cache with limited size (the limit may be infinite) where the oldest, by access date, version gets cleaned out, when the limit is reached. According to the distinction made above, we speak of *source histories* and *derived caches*.

Source Histories

A source history may contain three kinds of versions. *Busy* versions (mutable versions that may be altered by a user), *saved* versions (read-only versions), and *clones* (replicas of saved versions). All versions of a history carry the same name, that means the name is in fact an attribute of the history which is inherited to all its versions. Conceptually, a version itself does not have a name attribute.

Busy versions are always stored as regular UNIX files. They may be modified by eg. an editor at any time. AtFS gives the possibility to store application defined attributes with busy versions additionally to the standard attributed stored with the UNIX file in the file system.

Saved versions come into being by saving a copy of an existing busy version. The copy is stored in an AtFS-internal part of the file system. This part is not to be accessed via regular UNIX file system operations by the user. Saved versions are read only and their contents may under no circumstances be modified. The attributes of a saved version are also stored in the AtFS internal part of the file system together with the additional attributes for busy versions mentioned above.

Clones are replicas of existing saved versions. A clone is conceptually the same object with the same version identifier as the corresponding saved version. Clones are also stored as regular UNIX files. They are protected against modification, as their contents may not be altered. Of course AtFS has no means to prevent users from changing the UNIX protection and modify the contents of a clone.

When modifying a clone by means of the UNIX file system, without control by AtFS, a user implicitly changes the clone to a busy version. This is not the correct way, but AtFS is able to deal with that. This case is considered as a checkout of an old version to make this one basis for further development. The saved version, the clone was cloned from, is the predecessor of the new implicitly created busy version.

Remember the fact, that all versions in a history have the same name, which is true also for busy versions and clones, stored as regular UNIX files. Due to the fact that the UNIX file system does not allow equally named files in one directory, one directory may contain maximal one clone or one busy version of each history.

Saved versions in a source history evolve in a tree like manner. When a new saved version is created, its predecessor version might be an old one rather than the most recently saved version. In this case, a new line of development branches off an old version.

Version numbers for saved versions and for busy versions come from different domains. Saved versions carry positive integers as version numbers, while version numbers for busy versions are negative. Saving a busy version creates a copy of the busy version as new saved version with new version number. The version numbering for saved versions follows the time axis, that is, when a new version is created, it gets a version number that is the successor of the version number of the most recently saved version. Each saved version has an attribute denoting its predecessor version, the version it was derived from.

Figure 2 shows a version evolution tree with busy versions and clones. The arrows between the saved versions visualize the physical evolution history. Each of the two busy versions are candidates for being saved. Busy version #-1 evolved by manipulating saved version #3 and busy version #-2 bases in saved version #5. When saving a copy of #-1, the new saved version (#6) will be successor of #3, while when saving #-2 it will be successor of #5.

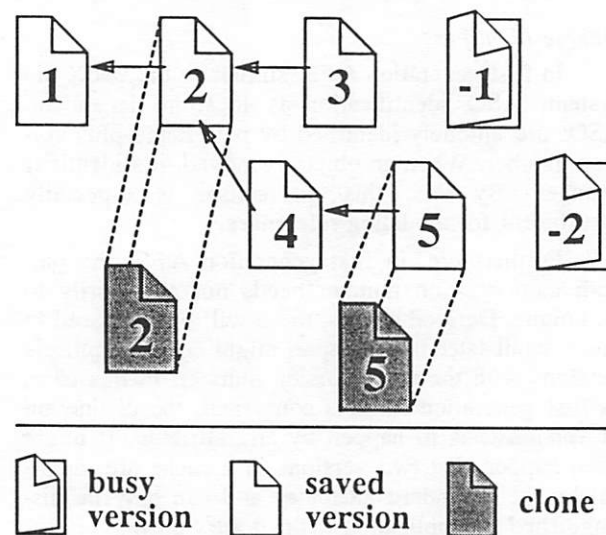


Figure 2: Source History

Derived Caches

Derived caches are intended to hold derived versions, versions that can be reproduced at any time. Like in source histories, each version in a derived cache carries the same name. Typically, all versions in a derived cache evolve from the same (set of) source history(ies). The difference between the versions might be, that they evolved from different source versions, by different derivation tools, or in a different derivation context (different options). The number of versions stored in a derived cache may be limited by the application. If the limit is reached, for each new version stored in the cache, the oldest version (the version that has not been accessed for the longest time) gets cleaned out.

Derived versions can also carry application defined attributes.

Typical examples for a source history and a derived cache would be a C-module `foo.c` and the corresponding object code `foo.o`. Let's assume, the source history of `foo.c` consists of three saved versions (1,2,3) and a busy version(-1). Let's further assume, `foo.c` is part of a bigger system, where one release is already shipped to some customers, a new release is just to be configured and further development happens. The already shipped release contains `foo.c#2` compiled with the `-O` (code optimizing) option, the release currently to be constructed contains `foo.c#3` with the same compile option. In the current development the developer of `foo.c` tests his new developments using a compiled version of the busy version produced by using the `-g` (generate debugger information) option and another developer test his stuff by invoking the most recent saved version (`foo.c#3`) compiled with `-g`. This typical situation has four different versions of `foo.o` that evolved from three source versions (2,3,-1) with two different derivation contexts (`-O,-g`).

A derived cache may also contain complex programs configured from a set of other derived versions, which is another typical example. However AtFS does not enforce any concept with derived caches. An application may put any file into a derived cache.

Attributes

The following sections deal with AtFS objects that are not ASOs. Figure 3 shows the root of the AtFS type tree and makes the type hierarchy complete.

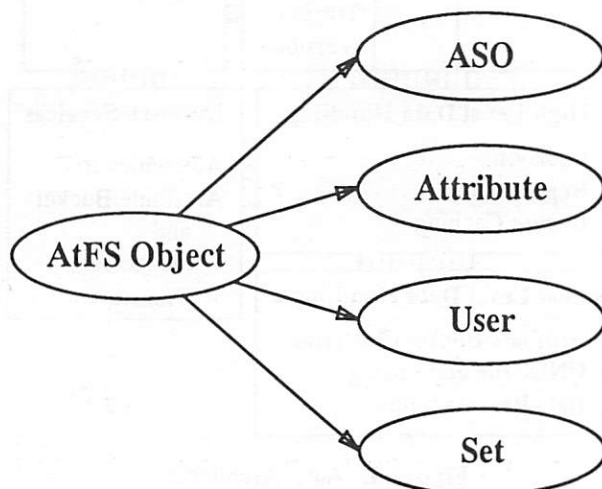


Figure 3: AtFS basic types

In first generation AtFS, attributes simply were strings of the form *name=value*. That means, the value of an attribute was restricted to strings. Second generation AtFS introduces typed and structured attributes. An attribute consists of a name and a list of values each of which can be of a predefined type. Predefined types are

- string – byte structured strings
- integer – long integers.
- real – real values
- user – an AtFS User structure containing user name, realm and user ID (described later)
- ASO handle – as described above (can be used to model references)
- binary – binary data controlled by the application.

For all attribute value types, except for binary values, AtFS does a proper byte swapping and alignment for exchange of the values between different machines in the network. Binary values are intended to hold application controlled structures whose contents AtFS is not aware of. For binary attribute values, the application has to perform byte swapping and alignment itself, if necessary.

An application may associate a routine with each attribute that gets activated by AtFS when the value of the attribute changes. This mechanism helps to implement sophisticated models for triggering dedicated actions on specified events. A routine associated with the standard attribute “date of last status change” will be called if *any* modification, either of the contents or any attribute, is performed on an ASO.

Users and Authorization

As AtFS maintains histories spread over a network, it needs a network user concept. The owner of a history has to be identified as the same when looking to a history locally or from a remote machine. The system shall be able to identify `andy@coma` as the same user as `andy@blurp` when this is in fact the same person logged in from different machines in a workstation network.

The AtFS network user concept uses the *Kerberos* [Steiner1988a] authentication mechanism. Kerberos allows the identification of users in a *realm*. A realm may be a workstation network or all hosts belonging to one site. In our former example, `andy's` user ID would be `Andreas_Lampen@cs.tu-berlin.de` an identification, that is world wide unique.

Beside the *authentication* by use of Kerberos, the *authorization* mechanism of AtFS is similar to that of the UNIX file system. In fact it has to live peacefully together with the UNIX file system because part of AtFS are regular UNIX files and access to these file is only possible if the file system

allows this.

In AtFS, each directory and each history has an owner and each version has an author. In the case of busy versions and clones, the author is the UNIX owner of the corresponding file.

History permissions can be set to allow reading the history, adding or deleting attributes and locking the history for updating. These permissions can be granted to the owner of the history, members of the owner's group or the rest of the world. Permissions are stored, similar as in the UNIX file system, as a bitfield.

Versions have permissions for reading, writing (resp. adding/deleting attributes) and executing. This is very close to the model in the UNIX file system with the addition that write permission also allows addition and deletion of attributes. These permissions can be given to the author, members of the author's group and the rest of the world.

Attributes that are associated to an ASO have read and write permissions. These permissions may be given to the owner of a directory or a history and members of the owner's group. For version attributes, read and write permissions can additionally be given to the author of the version and members of the author's group. For each attribute read and write permissions may additionally be granted for the rest of the world.

Sets

A very useful feature, that was already present in first generation AtFS is a simple retrieve interface for nonunique identification of ASOs by any attributes. The search space for a retrieve operation is a directory. A retrieve operation is restricted to just one kind of ASOs in a directory, either histories or versions.

The result of a retrieve operation is stored in a *set*. AtFS provides a certain number of routines on sets allowing the building of unions, differences and intersections of sets, as well as addition and deletion of single set elements. Complex retrieve operations can be implemented by performing multiple simple retrieve operations and combining the resulting sets of ASOs.

Second generation AtFS additionally features sets of attributes. The functions on sets of attributes are the same as these mentioned above, although the intention may be different. A set of attributes is needed as input for any retrieve operation and so often needs to be constructed manually from scratch. On the other hand, a set of attributes can also be result of a function returning all attributes of an ASO.

A look at the implementation

The implementation of first generation AtFS is in use for about two years now. As a part of the shape-toolkit it is public available and is used in at least a handful projects regularly. It has also been used as basis for an experimental system supporting communication between developers in large software engineering projects [Ecker1990a], developed at the TU Berlin.

The second generation requires a totally new implementation. We defined a new interface meeting the extended notion of ASO and introducing goodies like dynamic exchange of the delta procedure and dynamic cache space limitation. As the type hierarchy in this paper suggests, the new interface was designed in an object oriented manner.

First generation AtFS is implemented as an extension to the UNIX file system. The same is true for the new implementation. It is important, that no modifications to the UNIX file system (kernel modifications) are necessary. AtFS comes as a library simply to be linked to an application.

Due to limited space in this paper but also, to be honest, due to lack of experience with the reimplement, we will in this section only discuss a few interesting topics concerning the realization rather than giving an exhaustive description.

The reimplement is only partly finished up to now, although the design is set. Figure 4 shows the basic architecture of the implementation of second generation AtFS.

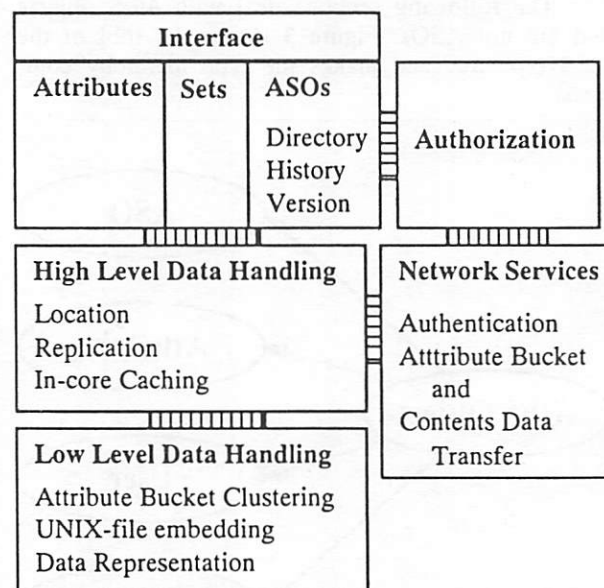


Figure 4: AtFS Architecture

Low Level Data Handling

Low level data handling deals with the physical storage and clustering of AtFS internal data. In this section, techniques for efficient access and space saving storage of AtFS data will be discussed. Another issue in this section is the embedding of information stored in the UNIX file system into AtFS's domain.

AtFS's data are stored in a reserved part of the UNIX file system, namely subdirectories named AtFS, which are potentially present in each directory. Conceptually, all data belonging to a history are stored together in one subdirectory. This concerns all contents data for saved and cached versions, the file system attributes for these, and application defined attributes for *all* versions of a history. Contents data and file system attributes of busy versions are not stored together with the other history data in the subdirectory, but rather as regular files in the file system.

We speak of all history data mentioned above: saved, cached, and busy versions and all their attributes, as the history's *master data*. The master data for a history are kept redundancy free. This is the reason why no information about busy versions is replicated in the subdirectory. Clones and their attributes do not belong to a history's master data as they are just replicas of saved versions.

All attributes AtFS stores for one version are packed together in an *attribute bucket*. In first generation AtFS all attribute buckets for one history were clustered together in one archive file. This came from the tradition of version control systems like SCCS and RCS. Unfortunately this technique implies some performance problems. Experiences show, that most accesses to a history address either a busy version or the last saved version. Hence, it is much more reasonable, to cluster together the attribute buckets of the busy versions and the most recently saved versions of all histories in a directory.

We haven't found our final concept for attribute bucket clustering in second generation AtFS yet. We rather plan to implement different clustering techniques and make up our final decision after some performance measurements.

The contents data of all versions in a history is stored in a space efficient manner. This is either done by storing deltas (differences between files) or compressing the contents data. Delta technique is applicable only to saved versions of a source history. AtFS provides different procedures for generating deltas. Additionally, each application may introduce it's own pair of delta-generating/regenerating procedures. Basic equipment of AtFS are GNU diff/ed and an own delta technique based on Tichy's string to string correction algorithm [Obst1987a].

The contents of cached versions is stored in a compressed form. Again, the application may introduce it's own function for compressing/uncompressing. The mechanism for introducing application defined data conversion procedures also opens the possibility to provide a *crypt* algorithm for encrypting the data. Compressing or encrypting the data is also applicable for source versions instead of constructing deltas.

The application drives at run time, which compression or delta algorithm is to be used. When a version is stored using a non-standard delta or compression algorithm, this algorithm has also to be present when trying to restore the concerned version. Internally, AtFS associates each version with a delta or compression type *number*. For non-standard procedures, the application has to keep the mapping between delta/compression type number and the appropriate procedure pair consistent. An application writer should always be aware that other applications will not be able to restore the data of the corresponding version without the appropriate procedure. This however does not affect the information about the version, the attributes.

High Level Data Handling and Network Support

High level data handling concerns the management of histories including data replication strategies and in-core caching of data. The coherence of a history has to be maintained, busy versions and clones, even if they are located on remote machines, have to be kept in touch with their history.

Local to each directory, UNIX files are mapped to histories in the AtFS subdirectory. The mapping bases on the name of the file. Renaming a file by means of the UNIX file system causes loss of the information about membership to a history.

The history in the subdirectory may either be a real one, containing the history's master data, or the *history link* pointing to some location in the network where the real history lies. History links are maintained in the same way as ASO handles, containing a unique identifier of the history and a location hint. A history link is needed for each clone and each busy version, located somewhere but not in the directory right "above" the history data. When a history's master data are moved to another place, AtFS leaves the information, where the history has been moved to, behind. This also happens in form of a history link.

A source history may be spread over multiple machines in the network. As all saved versions in a source history are stored together in one subdirectory, only clones and busy versions may be located on another machine than the history's *home*. In the case of a distributed history, a replica of the corresponding attribute bucket is stored together with each remote clone and busy version. AtFS keeps

history master data an attribute replica consistent by trying to perform an update every time, when a change to the replicated attributes or the the corresponding master data was made.

Change propagation as described above may not always be possible. If eg., the history's home machine is down, while an attribute of a remote busy version is changed, AtFS has to deal with inconsistencies. In this case, the replicated attributes are considered to be the real ones and updating is deferred. Deferred updating might also be desired in a wide area network where contacting the history's home machine on each access is too expensive. To meet all these needs, AtFS supports different updating strategies for replicated data.

On run time, AtFS caches history data read from disk in core. Each application may set a space limit for the cached data. Cached data and the corresponding data on disk are always kept consistent. Modifications to cached data get written to disk immediately. AtFS performs reader/writer locking for applications on whole attribute buckets. While an attribute bucket is updated on disk, all other applications trying to access the attribute bucket are deferred. For access to contents data, AtFS provides nonexclusive, as in the UNIX file system, and exclusive open operations.

Conclusion

It is sometimes hard to motivate, what a system like AtFS is really needed for. AtFS has to be seen as a certain level of abstraction in a bigger whole. It's task is it, to build a bridge between conventional file based tools and applications, that need more sophisticated data management support.

In it's role as part of the kernel of a generic software engineering environment, AtFS comes along together with a system for describing a type hierarchy for software objects, and an object oriented command interpreter.

The type system allows to describe *classes* of software objects that are more specific than just attributed software objects. For example, all ASOs containing C-language source code would be associated with properties that are characteristic for C code modules. The type system is defined in an object oriented specification language featuring multiple inheritance, generic classes, method overloading, dynamic identification, and schema evolution. The so described *typed software objects* are stored by means of AtFS. Their properties (attributes and methods) are mapped to application defined attributes in AtFS.

The type system is enacted by *OShell*, an object oriented command interpreter resembling the UNIX shell. OShell allows the user to send objects messages that trigger invocations of object specific methods. OShell maintains a type attribute for each

ASO mapping it to a specific class in the type system. The class description gives semantics to the ASO's application defined attributes either representing class attributes or methods. An exhaustive description of the type system and OShell can be found in a companion paper to this one following right after in these proceedings [Mahler1991a].

The type system has a lot commonalities with a schema definition language for an object oriented database. However, compared to databases, the schema information is much less essential in our model. Even if the schema information gets lost totally, the data still remains accessible in a quite convenient way by means of AtFS. This makes it especially attractive for network distributed applications, where, due to network problems, not always accurate schema information may be available. We think, that this approach provides the right mixture of robustness and functionality.

Availability

First generation AtFS was distributed together with the shape-toolkit over *Usenet* (in *comp.sources.unix*) about two years ago. In spring 1990, a new release of the shape-toolkit was released. The new release is available via anonymous ftp on several servers.

Second generation AtFS, a totally new implementation, is only partly finished up to now. We would have liked to include some performance measurements into this paper, but this will take us at least a few more months. As performance is a most important issue with file systems, we want to perform the design of the performance critical parts very carefully. And this takes a while.

Acknowledgement

This work is part of the STONE project. STONE – a Structured and Open Environment – is a research project supported by the German Ministry of Research and Technology (BMFT) under grant ITS-8902E8.

References

- Banerjee1987a. Jay Banerjee, Won Kim, Hyoungh-Joo Kim, and Henry F. Korth, "Semantics and Implementation of Schema Evolution on Object-Oriented Databases," *Proceedings of the Annual SIGMOD Conference* 16, 3 pp. 311-322 ACM, (May 1987).
- Bretl1989a. Robert Bretl, David Maier, Allen Otis, Jason Penney, Bruce Schuchardt, Jacob Stein, E. Harold Williams, and Monty Williams, "The GemStone Data Management System," pp. 283-308 in *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison Wesley, New York, New York, USA (1989).

- Clemm1986a. Geoffrey M. Clemm, *The Odin System: An Object Manager for Extensible Software Environments*, The University of Colorado, CU-CS-314-86, Boulder, Colorado, USA (February 1986).
- Dittrich1986a. Klaus R. Dittrich, Willi Gotthard, and Peter C. Lockemann, "DAMOKLES - A Database System for Software Engineering Environments," *Lecture Notes in Computer Science* 244 pp. 351-371 Springer Verlag, (June 1986).
- Ecker1990a. Markus Ecker and Christian Focks, *Entwurf und Implementierung einer Werkzeugs zur Unterstützung der Kooperation in Softwareentwicklungsprojekten*, Technische Universität Berlin, Diplomarbeit, Berlin, Germany (September 1990).
- Haberman1986a. A. Nico Haberman and David Notkin, "Gandalf: Software Development Environments," *IEEE Transactions on Software Engineering* 12, 12 pp. 1117-1128 IEEE Computer Society, (December 1986).
- Khoshafian1986a. Setrag N. Khoshafian and George P. Copeland, "Object Identity," *SIGPLAN Notices* 21, 11 pp. 406-416 ACM, (November 1986).
- Lampen1988a. Andreas Lampen and Axel Mahler, "An Object Base for Attributed Software Objects," *Proceedings of the Autumn 1988 EUUG Conference*, pp. 95-106 European Unix systems User Group, (October 1988).
- Mahler1988a. Axel Mahler and Andreas Lampen, "An Integrated Toolset for Engineering Software Configurations," *SIGSOFT Software Engineering Notes* 13, 5 pp. 191-200 ACM, (November 1988).
- Mahler1991a. Axel Mahler, "Organizing Tools in a Uniform Environment Framework," *Proceedings of USENIX Technical Conference*, USENIX Association, (January, 1991).
- Mogul1986a. Jeffrey C. Mogul, *Representing Information about Files*, PhD Thesis, Department of Computer Science, Stanford University, Report No. STAN-CS-86-1103, Stanford, California, USA (March 1986).
- Nestor1986a. John R. Nestor, "Toward a Persistent Object Base," *Lecture Notes in Computer Science* 244 pp. 372-394 Springer Verlag, (June 1986).
- Obst1987a. Wolfgang Obst, "Delta Technique and String-to-String Correction," *Lecture Notes in Computer Science*, pp. 69-73 Springer Verlag, (September 1987).
- PCTE1988a. PCTE, *PCTE - A Basis for a Portable Common Tool Environment*, Commission of the European Communities, Brussels, Belgium (November 1988). Functional Specification, Version 1.5
- Skarra1987a. Andrea H. Skarra and Stanley B. Zdonik, "Type Evolution in an Object-Oriented Database," pp. 393-415 in *Research Directions in Object-Oriented Programming*, MIT Press series in computer systems, Cambridge, Massachusetts, USA (1987).
- Steiner1988a. Jennifer G. Steiner, Clifford Neumann, and Jeffrey I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *Proceedings of USENIX Technical Conference* 16, 3 pp. 191-202 USENIX Association, (January 1988).
- Tichy1988a. Walter F. Tichy, "Tools for Software Configuration Management," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1-20 German Chapter of the ACM, (January 1988).
- Wileden1988a. Jack C. Wileden, Alex L. Wolf, Charles D. Fisher, and Peri L. Tarr, "PGRA-PHITE: An Experiment in Persistent Typed Object Management," *SIGSOFT Software Engineering Notes* 13, 5 pp. 130-142 ACM, (November 1988).

Andreas "Andy" Lampen is researcher at the Technische Universität Berlin, Germany. He's been working in two research projects dealing with the construction of software engineering environments. His special interests concern software configuration management and distributed data management in software engineering environments.



Andy received his Diplom (MS degree) in computer science from Tech. Univ. Berlin in 1985. His electronic mail address is andy@coma.cs.tu-berlin.de. Reach him via paper mail at TU Berlin, Secr. FR 5-6, Franklinstr. 28/29, D-1000 Berlin 10, Germany.

Organizing Tools in a Uniform Environment Framework

Axel Mahler – Technische Universität Berlin

ABSTRACT

The UNIX tool-collection provides an excellent platform for software development. The mere existence of literally hundreds of different tools for all sorts of activities makes the UNIX toolbox one of the most effective software development environments around. However, all these tools are independent from each other, solitary in design, and thus are poorly integrated. It has often been argued that the vast number of tools, all with different sorts of options, and different command sets makes UNIX sometimes hard to use - even for experienced programmers. Well, it's not exactly *hard* to use the UNIX environment, it's more that the potential of the environment is often not well enough developed. Users tend to use only a fraction of the possible benefits that the toolsystem offers.

The concept of an object oriented command-interface provides a rather simple but powerful solution for this problem. It allows to integrate about any UNIX tool into a coherent software development environment¹ framework. While retaining the familiar concept of UNIX-commands, the system allows to offer *functionality*, implemented by tools or tool-combinations, in a uniform and consistent way at the user interface level. This is achieved by enhancing the notion of *files* to that of *software objects*. Software objects are different from files in that they have *type* as a basic property. Type-specific functionality is centered around these software objects in an object oriented fashion. The user accesses software objects and related tool functionality, represented as methods, through the object-shell (*OShell*), a command interpreter that makes consequent use of the underlying type system. The object-shell together with a powerful type definition language provide the potential to further develop the vast tool capabilities of the UNIX system by organizing functionality around classes of software objects with varying degrees of specialization. The possibility to build class-based abstractions for a concrete environment allows to integrate characteristics of the work *process* with the environment itself.

Background

Integrated software development support environments (SDEs) are vital infrastructure for professional, especially large scale software development projects. Highly integrated, specialized but unfortunately *closed* toolsystems have the disadvantage that they can't make proper use of external tools. While offering good support for specialized tasks these systems are inherently unflexible and often deprive the programmer from some or all of his most cherished and most effectively used tools. With the development of an object-oriented command interpreter we are able to provide a framework for integrating unrelated UNIX tools on top of an enriched file system into a moderately high integrated SDE. The described work is unique in its consequent use of object oriented principles and techniques that are (almost) seamlessly combined with the well-known concepts of the UNIX

environment.

The described work originates from the development of the *shape toolkit*, an integrated set of programs for version control, and software configuration management for UNIX[1]. The central idea of the shapetools system was the *attributed software object* (ASO), providing a uniform abstraction for UNIX files, and versions of these files. The abstraction was implemented by the *Attributed File System* (AtFS), an enhancement of the standard file system. For more details on AtFS, see the companion paper *Advancing Files to Attributed Software Objects* Lampen Advancing Files in this volume.

Attributed software objects are basically made up of contents, and an associated *set of attributes*. Among these attributes are those that are inherited from the UNIX file system (name, owner, size etc.) as well as any number of arbitrary, so called *user defined attributes*. All attributes have the form *name=string*, with no limit on the size of *string*. We designed the attribution mechanism this way because we weren't sure, which or how many attributes would exactly be necessary for our version control and configuration management system. One of our

¹Although this work is mostly concerned with building software development environments, the described approach is also applicable to the UNIX working environment in general.

objectives was to be able to configure complex software systems according to required properties that system components shall satisfy in order to be eligible for a planned configuration. The concept was that all component versions of a software system should be annotated with attributes describing the object [2] (such as "is tested", "needs polishing", "has been shipped" etc.). However, an unsolved problem of this concept was that all attributes that hadn't a hardwired meaning had to be manually attached to the objects².

While further improving our toolkit and gaining experience using it, we gradually understood that configuration management and version control are not just necessary software engineering tools but an elementary technical platform for coordinating teamwork within software development projects. This enhanced understanding partially shifted our focus of interest towards software development environments. The ASO object-base abstraction, and the configuration management system built on top of it seemed to be a good starting point to build a framework for the construction of integrated SDEs from existing tools. The possibility to attach attributes to software objects promised to be an excellent basis for passing information between developers, tools, and the environment.

The construction of comprehensive, integrated, yet open and arbitrarily extensible SDEs from existing tools is one of the hottest research and development issues in software engineering support today. The idea of giant, monolithic, tightly coupled, and closed SDEs, addressing all (present and future) software development activities and techniques has widely been dropped. New studies emphasize the importance of incorporating already existing tools into integrated environments ("tool re-use").

A number of research projects have significantly influenced our work, or go into similar directions. Currently, there are two mainstream approaches to environment construction that are oriented towards re-use of existing tools. We'd like to refer to these approaches as *object-based* and *tool-based* integration frameworks.

Object-based frameworks

Generally, object-based approaches to environment construction try to capture the specifics of the software development process by classifying the pieces of information evolving in a project, and describing related *behavior* of these information objects. We will refer to them as *software objects*. This basic concept of environment integration has been described by Osterweil in [3].

²This is done by using the version control commands of the toolkit, which offer a complete interface to the attribution facility from the UNIX command level.

The object oriented paradigm is used to specify external properties of software objects, and to provide for formally defined access protocols that must be followed when these objects are manipulated. Use of type inheritance makes it possible to specify general functionality on an abstract level (*abstract superclasses*), making the functionality available for use/reuse in application specific class definitions (*specialized classes*). The specified functionality is turned into action by corresponding *process programs*, formal (and executable) descriptions of particular software engineering activities, enacted by a process program interpreter. These process programs are the *methods* associated with a given software object class[4, 5].

Budd describes in [6] the design of an object oriented command interpreter for UNIX, considering files as objects, and introducing a command syntax that resembles Smalltalk. Many recent research projects have adopted the notion of *Software Object* as central for environment construction, and extension mechanisms[7-9]. Clemm's Odin system represents a very sophisticated approach to object management, tool-, and process integration within the UNIX environment. It consists of a specification language for describing software object properties, and a request language to access the objects. Odin is built atop a concept that distinguishes *source-* and *derived* objects, and is especially tailored to manage the most complex software derivation processes, such as compiler generation. Unfortunately, the Odin system is a bit awkward to use which accounts for the fact that Odin isn't by far as widely known as it should be.

There are a number of other object-based environment projects that contributed substantial impulses for the construction of open environments within integration frameworks. Without the intention to discriminate others we'd like to mention Sun's NSE[9], and the object management system of PCTE[10]. The work described in this paper should also be classified in the category of object-based environment projects.

Tool-based frameworks

In contrast to the previous approach, tool-based integration frameworks concentrate on adjustment and tuning of unrelated tools in order to compose ever more powerful, integrated tools. The basic principle is to represent tools as services that are advertised within the environment framework and thus can be used by other tools to implement their service. The most prominent system taking this approach is Reiss' FIELD environment[11]. FIELD integrates existing tools under a common graphical user-interface. Tools communicate via messages, representing information and commands intended for other tools. Messages are sent to a central environment message server which dispatches the messages

to all tools possibly interested in them (*selective broadcasting*). The technique permits to integrate unrelated tools by adding a *message interface* to them. While this approach makes it possible to create moderately highly integrated environments from existing tools without a great deal of work, it requires some modifications to the tools' source code.

HP's Softbench[12] is based on the same tool communication concept but goes one step further in supporting systematic tool integration. The *Encapsulator*[13] tool permits to fully integrate any "standard" (i.e. tty-oriented) UNIX-tool into the Softbench environment, without the modification of a single line of code. The idea is to write a tool encapsulation program in an *Encapsulator description language* (EDL) that provides the necessary "glue" to transform a raw tool into a consistently behaving part of the environment. EDL allows to describe a graphical front-end for a tool, to define its message interface, and to insulate the raw tool interface from the outside.

Another project taking a similar approach is the Eureka Software Factory's *Software Bus*, into which so called *service components* are conceptually "plugged in"[14]. The services offered by the service components can be accessed from *user interaction components* that are likewise plugged into the software bus.

Tool-based integration frameworks are particularly well suited for networked environments, as message-based communication is the fundamental integration technology anyway. A problem is the somewhat awkward specification of service-interfaces, which can make it kind of hard to use tools that are already integrated in an environment. Object-based systems have their strengths in a comparatively well structured, explicit specification of the objects' capabilities. In exchange for that, this type of environment needs relatively much additional support for taking advantage of network resources.

This little survey cannot claim to be complete in any way, as SDE projects are too numerous to be covered in full breadth.

Towards a Smarter Command Interface

When we came up with the idea of attributed software objects, we felt that it was a very simple, orthogonal, and powerful concept. We simply hadn't the idea yet, how to make ultimate use of the facility. As we got more experienced with our own toolkit, we began to play around with attributes in a variety of ways: we used them, for example, to attach informative annotations to objects, or to store complex commands (such as typesetting command pipelines) that are applicable to certain objects. This very paper for example has an attribute:

```
print='soelim Usenix.ms | \
refer -e -P | troff -Tps -mux | \
tps | lpr -Plw''
```

The *print*-attribute³ is used to conveniently format troff documents. However, there was one thing that left us unsatisfied: use of non-hardwired (user-defined) attributes was unsystematic; it was not based on any sort of protocol defining how to interpret or use an attribute, and thus meaningless. We still had the problem of how to *systematically*, or more precisely *automatically*, associate meaningful attributes to our software objects, so that for example the configuration tool may use them for selecting configurations with specified properties.

In response to this problem, it kind of suggested itself, to introduce some sort of *class* notion for software objects that would permit to define sets of attributes characteristic for certain kinds of software objects. So, all instances of software objects of a certain kind would share the same set of attribute slots. It was also desirable to use the attribution facility to associate functional properties with the objects. This would be the precondition for using attributes and functions in a well defined manner (i.e. tools/methods could have a common idea about what the attributes mean).

An object-oriented inspiration

We realized that we were just about to introduce a *type system* for UNIX software objects. However, a type system only makes sense if it is consequently obeyed, well respected among all tools and properly enforced. We assessed that it would be impossible to implement this principle without getting Shell under control. A well behaving command interpreter as main user-interface to software objects appeared to be a key-component. Budd described in [6] the idea, and a prototypical realization of an object-oriented command interpreter that tried to combine concepts from Smalltalk and the UNIX shell. He gave a good description of the benefits and the problems with this approach. Among the benefits are ease-of-use, and relieving the (tool-) name space congestion. A major problem with this approach is UNIX' inherently procedural tool philosophy: all tools are directly accessing and modifying the contents (private state) of the software objects. This seems to be the anti-thesis of object-oriented principles - or is it?

Budd's *osh* idea was most inspiring for us, because our concept of attributed software objects was the natural solution for a couple of problems that Budd had with his concept. In particular the

³This attribute can be used to print the paper with `eval 'pattr -uda print Usenix.ms'.` "Pattr" writes the value of a specified attribute to standard output; the actual print-command is realized as a shell-function.

problem to reconcile some of UNIX' most important and useful features, namely *pipes* and *pattern-matching*, with the object-oriented paradigm is only feasible when class information is a genuine property of objects.

With the design of OShell, we wanted to make sure that pipes and regular expressions were preserved, while at the same time the command line syntax was kept as simple and clear as known to the UNIX user. The resulting command interpreter offers a basic command line syntax that is very similar to that of the regular UNIX shell, providing regular expressions for object names, and a concept of *type-safe pipelines*. While offering control structures and other nifty features, the basic structure of an OShell command is as follows:

`<message> <recipients> , <arguments>`

At first glance this structure will look quite familiar to shell-users, where the basic command structure is:

`<command> <arguments>`

with `<arguments>` normally made up of a couple of file names and option-switches to the command. The OShell command syntax might, on the other hand, look a bit surprising for object oriented people, who tend to expect an object reference (recipient) at the leftmost position and the message coming after that, perhaps separated by a dot. Well, who ever said that object oriented commands *have to* look this way? The important thing is, that conceptually a *message* is sent to one or more *recipient objects*. As a matter of fact, the "traditional" object-oriented notation has some drawbacks that make it unsuitable for command-line oriented interfaces. First, one has to know all objects in order to send one of them a message. This requires that all objects be permanently displayed somehow. Second, it is not straightforward to send the same message to several objects at a time. Our idea was, for example, to send a *compile* message to all component objects of a system, and have each object compile itself orderly, perhaps with instance specific compile switches or different compilers. This shouldn't be more complicated than issuing something like

`compile *.mod`

The regular expression is substituted by all objects' names that match the specified pattern. OShell then looks up the appropriate *compile* method for each object, and activates the actual compilations. While this is a rather trivial⁴ example, it displays the power of abstraction that an object-based system organization can provide. The objects matched by `*.mod` need not to be of any particular programming language, in fact they need not to be

programming language modules at all - maybe some of them are documentation objects that are formatted and sent to a printer in response to the *compile* message. The sole requirement would be that all objects respond to *compile*.

A drawback of OShell's syntax is the somewhat odd separation of the argument list with a comma. This was necessary because there is no way to lexically tell object identifiers from arguments, which are both represented as strings. But the possibility to use regular expressions in place of `<recipients>` and `<arguments>` outweighs this oddity by far. A more detailed description of OShell's design and implementation will be given later.

Organizing Objects in a Type-System

Before an object oriented command interpreter can use the class information associated with objects, this information must be created somehow. In his implementation of *osh*, Budd used a dedicated subdirectory, where he kept manually written class objects. This approach is only feasible for an experimental prototype with a small number of classes, defined by the designer of the system himself. For a more ambitious enterprise, like the described OShell system, we need a proper definition facility for software object types.

For this purpose, we developed *CHieF* (Class Hierarchy definition Facility), a language to define object oriented class type hierarchies. Concepts and language constructs of the class definition language were chosen in accordance to the access principles of the object shell. The language reflects all the concepts that the object system will later be able to display. The OShell will act as an agent that a user employs to explore an existing object base. The following sections will discuss the concepts, and applications of the induced type system quite thoroughly. A number of essential features of OShell will be anticipated in this discussion, so that they can be handled briefly in the section describing the OShell.

The class definition language

The CHieF class definition language features multiple inheritance, generic classes, method overloading, and easy schema modification. The specification mechanism allows to easily describe the properties of the various software objects that are encountered in the software development process, such as relations to other software objects, or dependencies. It also encourages to view the objects in a system as instances of abstract data-types that are manipulated in a well defined manner, maintaining some sort of invariant (it does not provide any means to *enforce* invariants though).

Based on first experience with an early version of CHieF, the language definition has recently been revised. The improvements are mostly concerned

⁴In real life one would start a build process by sending a build message to a systemmodel object[15].

with better support for organizing the namespace for class-types (*domains*), better facilities for the description of method side-effects, and more provisions for user-interface agents (see discussion below).

```

Domain: general
Class Users
  "Users represents a directory of
  all user-accounts on the
  current host or network."
Inherits: Directory
Class Features:
  Int cardinality=0; -- at most one
Features:
  list 0 {
    "Create a listing of all
    known user-accounts as
    standard-output object."
    Effects: None
    Plugs: List[User] Out;
    Method:
      Command ("listusers.sh")
  }
End -- of Users class-definition

```

Figure 1: Sample CHieF class-definition

The overall design rationale for the type definition language was to provide a sound basis for incorporating tools into integrated software development environments. It should offer enough flexibility to adequately handle general purpose tools (as found on most UNIX systems), as well as more specialized tools that might be purchased from an external source (such as database application development tools). Type systems that are built with CHieF should be straightforward, and rapidly adaptable in order to respond effectively to changing requirements for a development support environment.

The class-definition in Figure 1 displays a number of CHieF's language features. Elements displayed in sans-serif are part of the language, *italics* are used to indicate user-defined parts; "--" marks a comment. A CHieF module typically consists of a *type domain* specification, and a list of actual class-definitions. A type domain is a named scope in which type names are visible, and into which a defined name will be entered. The domain specification in the example indicates that the type name of class *Users* shall belong to the outermost domain *general*.

A *class-definition* consists of a class name (e.g. *Users*), a piece of documentation for the class, a heritage clause, and a list of *features*⁵. Features are either *attributes* or *functions*. Attributes are named slots that can hold a value of a specified type. These can be simple types, such as *Int* or *String*, or class types. Class type values are represented as references to the corresponding objects. Attributes

may be declared *class features*, in which case one value is shared among all instances of a class. Functions are always class features. They are merely specified, i.e. the interface and the possible effects of a method-execution are described. A function specification can be *virtual*, or contain a reference to an actual implementation, called a *method*. The concept of virtual functions is taken from C++[17] and serves to specify protocol for abstract classes without providing an implementation.

Function specifications provide the interface between the object system and the tools that are organized around object classes. Unpolished tool action shall be encapsulated within method-implementations and thus be cultivated into controlled "behavior". A function specification consists of

- the function's name
- a list of parameters
- a piece of documentation
- a list of possible *side-effects*
- a *plugs-list*, and
- a *method-link*.

Side-effects and "plugs" are specified to provide hints for the OShell, regarding which objects might be produced or destroyed by a method execution, and which kind of data will be sent to standard-output/error, or will be expected from standard-input. Side-effects are mainly specified for better user-interface support (see below), and to give the system (OShell) a chance, to check newly created *files* - properly typed - into the object base. Each function specification may specify up to three *plugs*, *In*, *Out*, and *Err* which are placeholders for the standard I/O-channels that a method execution process will have. Undeclared plugs are implicitly declared *void*, in which case the corresponding file descriptors of the method execution process will be closed (or connected to */dev/null*). *Plugs* are needed to provide hints for ensuring the typesafety of pipelines, and redirections. You can think of *In*, *Out*, and *Err* as dynamic, transient objects that have proper type but exist only for the duration of a method execution.

Representing class-information as software objects

When a set of class definitions is compiled from a CHieF module, a class object (an attributed software object of class *Class*) is created for each defined class. The class objects contain a template for the attribute slots of the class' instances, and the create method. *Class features* are represented as attributes of the class object. A class object maintains references to the class' methods as attributes, pointing to the corresponding *Method* objects. Methods are implemented by normal programs or shell scripts adhering to a special calling convention.

⁵The term *feature* is stolen from Eiffel[16].

It is the responsibility of these method implementations, to map the possible raw effects of UNIX tool processes, such as *vi*, *awk*, or *cc* to the specified behavior of a method. Figure 2 depicts the object structure, including Class- and Method objects, that would result from the compilation of the sample class definition above.

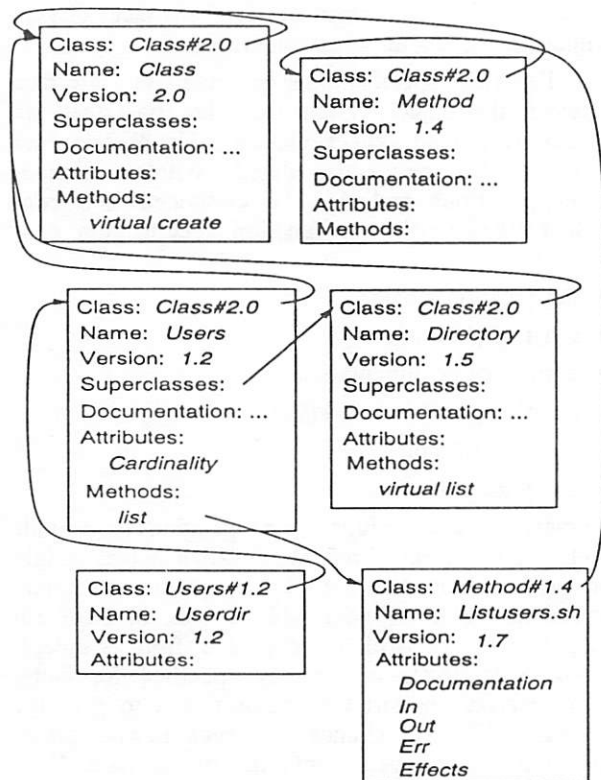


Figure 2: Objects and classes

Objects of a certain kind can be instantiated in a number of ways, such as *cloning* new from existing objects, or by sending a class object the message *create*, with the name of the instance to be created as parameter. Example:

```
clone proto.c, foo.c
create CSource, foo.c
```

This example illustrates, how OShell's commandline syntax provides a consistent tool activation interface, other than the traditional Shell commands, where ordering of parameters and options might be significant.

Type evolution

Support for type evolution in object oriented data management systems of any kind is an important aspect. Within the context of an open, adaptable SDE framework system, it is particularly important to cope with change of structural descriptions, such as class definitions. In [18] Zdonik discusses the importance of proper version control for regular objects as well as structural information (schemas)

within object bases, in order to ensure a consistent system behavior despite ever occurring changes.

The outlined class schema of OShell supports full version control (after all, the base system started as a version and configuration management system) for class definitions as well as method implementation objects. This provides for a very straightforward way of type evolution in the suggested object system (note the version attributes of the objects in figure 2).

The built-in AtFS version control system gives a system maintainer (the person who installs or improves a particular type system) the opportunity to test and debug a type system, before a stabilized version is finally frozen and made operational.

Setting up a type system

A type definition language provides the elementary means to build a system of useful abstractions that - in a way - formally models the system one is working with. Our approach to describing the system platform in an object-oriented way, was to strive for rather fine-grained functionality abstractions. Once they are identified, these abstractions can be defined as classes and be used to compose other, and more specialized classes from them. This approach relies heavily on multiple inheritance and implies a mix-in style of class-hierarchy construction. To give a taste of how the UNIX environment can be modeled with an object-oriented flavor, here are some of the most basic classes. For space considerations, only a brief, informal description of the classes is given instead of a CHieF definition:

Object is the root class of the entire class-hierarchy.

Its purpose is to model those properties that are common to all objects in the system. Some of its functions are *class*, *isKindOf*, *isMemberOf*, or *respondsTo* (very much as in Smalltalk).

Labeled is a very basic system class, representing *named* objects. This class represents all properties of general UNIX filesystem objects. All classes that can have instances must be descendants of Labeled. Although all objects are labeled, the class definition was kept separate from Object for conceptual reasons. Some of its functions are: *move* (move object to another location), *access* (check permission for desired action), and *changeProtection* (as owner, alter basic protection). Some of its attributes are *name*, *owner*, and *protection*.

Text represents all objects⁶ that share the (very common) property that their contents consists

⁶This formulation is used for convenience. Correctly speaking, a class does not represent all its instances, but rather properties that instances being "kind of" that class have.

entirely of printable characters. Some of its functions are the all-too-popular UNIX text tools: *grep*, *sort*, *tail*, a.s.o. This class allows to keep all those useful *filter* tools in an object oriented system. This can be illustrated by the CHIEF definition of the *grep* function:

```
grep (pattern, opt) {
  "Match the supplied pattern against
  all contents-lines of the
  receiving object and add
  them to the Out-object."
  Effects: None
  Plugs: Text In, Out;
  Method:
    Command ("grep.sh")
}
```

List is a descendant class of Text, representing objects consisting of lines that have all the same structure (such as a directory listing, a process table, or a password file). Some of its functions are: *line*, and *column*.

All these classes are abstract classes, not intended to have instances but rather providing elementary property abstractions that may be inherited by more specialized classes (see below). Other classes that fall into this category are *Binary*, *Directory*, *User*, *Process*, and *Executable*.

It requires a good deal of work to model the UNIX system and build workable abstractions for its resources. However, "finding the objects" and building abstractions for the user's point of view is also fun. One of our ideas was, for example, to define *abstract directories* that have - besides ordinary file-directories - specializations that contain other system objects, such as users, processes⁷, or network-nodes. We also liked the idea of being able to "edit" (or "open") fellow-users, eventually resulting in a *talk* (1) session.

A class model for software development

Until now, the examples of how the object modeling capabilities of the system are used have been rather trivial. It doesn't make a great deal of sense to replace the traditional view of the UNIX environment by an object-oriented one just for the fun of it (or does it?). The true benefit of such a system lies in more complex applications, such as software development environments (SDE). The purpose of SDEs is to support and guide the activities of developer teams, working on large systems. In theory the effect of an environment would be to relieve the members of a project team from having to be aware of complex procedures and behavior protocols that are essential for the functioning of a project. The environment ensures automatically that these protocols are followed. Programmer

productivity is increased because the programmer can better concentrate on the problem he/she is working on.

All software systems consist of a more or less large number of components, such as requirements, design- and specification documents, development objects (e.g. "modules" of all kinds), management plans, change-requests, memos, <what-have-you?>. All these objects represent valuable information that must be maintained according to its functional task within the development organization. There are also various relations among these objects (e.g. a change request might be tied to a new version of some source module). The AtFS object base, and the described object modeling system provide an appropriate basis to build environments that help to organize the development process. It is one of the basic properties of object based systems to define *protocol*, and *specific state* for objects. The protocol of a code module, for example, reflects its role within the development process and the discipline that has to be followed when the object is manipulated (e.g. updating cross reference lists or dependency information).

The task of environment construction consists of analyzing a particular development process, identifying the objects that are involved in the process, defining corresponding classes, and eventually implementing new methods. This is certainly easier said than done. However, the object system allows to organize all useful abstractions that have ever been found within an organization, in a well defined structure, thus making them available for easy re-use.

Currently, we are in the process of building a type system for an extensible, integrated software development environment. This environment will mainly serve as a test-field and proving-ground for our integration technology, and as such won't meet the requirements for real-world SDEs. The type system consists of a number of classes reflecting a special development principle: software configuration management. The development model assumes an analysis and design process that eventually leads to a *system-model*. A system-model is some sort of conceptual blueprint that identifies all the (anticipated) components of the system to be build. When the system-model is *instantiated* (this is different from instantiating a class!), all component objects of the system will be initially created. Once the object pool of a project is created, the individual component objects undergo numerous iterations of modification, resulting in versions and system releases.

The fact, that all system components - in our case C-language modules - are created by a call to a class method, permits to impose project-wide standards (such as a header style) for writing C-Modules. The cooperative authoring process is coordinated by a *locking-protocol* that guards software objects against concurrent updates. The class system makes

⁷More recent versions of UNIX do also have process-directories.

it straightforward to implement different locking policies according to different project needs (small projects are typically less formal and red-tapeish than large ones). Locking could for example be completely omitted or be connected to a complicated authorization scheme. Another aspect of the software objects' protocol is the way how modified versions of an object are entered into the project (*reserve-modify-test-propose-evaluate-publish*). For some more details on the development process see[1].

To give just a glimpse of how the outlined process is modeled with classes, we shall mention some of them:

SystemModel, and Variant represent structural descriptions of parts of the system, identifying all constituent parts of a (sub-)system. SystemModel is the central piece of information that guides system build-processes.

Source, Derived, and Derivable represent objects that are either manually created (by a human), or automatically derived by some process. The classes are needed to model the build process.

History is a property class (abstract) intended to be inherited by specialized source classes. It provides a *memory* for development objects that will be able to store and recall development stages. History also serves as transaction object, providing locking functions, protection, and a controlled release process for versions of development objects.

FormalLanguage, C, CModule, and CHeader represent some specializations of Source. FormalLanguage is an abstract class, representing all objects with contents that has a well defined syntax which might be validated.

A more thorough description of our class structure for an experimental SDE is beyond the scope of this paper. A somewhat more detailed description of this class hierarchy can be found in [15]. The sort of integration that is achieved, is to a lesser degree close tool interaction, but mainly a better integration of tools and the software process.

Type domains

Identifying the right classes that model your imagination of an intelligent support environment is - at least in our case - a longish, and iterative process (our project isn't primarily concerned with analyzing and class-modeling the UNIX-environment). However, we found a couple of basic system classes that are likely to be general enough to be used as building blocks for more specialized classes. In fact, we believe that quite a set of classes can be found and defined that will be useful for many type-system builders. The concept of *type domains* has been introduced into CHief to allow a partitioning of a given type-name space into general

and more problem specific parts. These parts can be maintained independently from each other. The most general domain might for example contain types of general nature, such as *Labeled*, *Text*, or *Process*. More specialized type domains, for example *development.general*, might contain class-types that model a certain concept of software development, such as *Source*, *Derived*, *Diagram*, *Variant*, or *SystemModel*.

Different environment maintainers (the people who configure a particular support environment, let's say for a project) may want to model the development infrastructure for their projects differently, but also want to use the *general* types. Setting up a new type domain, for example *myhomesmycastle.general* solves the problem. All type definitions from *general* will be available, while others that might be conflicting with definitions in *development.general*, can freely be introduced into the new domain. Type domains provide the possibility to separate more stable parts of a complex type system from experimental parts that are changed frequently. We consider the type-domain concept important because it helps to structure environment support according to the needs of different projects or teams within larger organizations. It is under consideration to associate type domains to name-servers that offer mappings from class-names to corresponding class-objects within networks. Network-wide type support for software environments would allow to maintain organization-wide standards for certain types of objects (e.g. documentation), while individual projects or teams can still develop their own specific type systems. This will allow different teams in an organization to follow, for example customized conventions for design transactions (e.g. locking policies) that suit their communication habits best. However, this idea is rather unelaborated to date.

The Object Shell

The object shell (*OShell*) is the operational component that allows to explore a given object system. *OShell* is the agent that gives (or denies) a user access to the objects and the objects' attributes, and makes sure that the objects are only affected by their own functions. Other than traditional shell-programs *OShell* provides also access to *versions* of objects. In fact, versions are a pervasive concept in our environment framework, that affected even *OShell*'s command-line syntax.

The concept of *object names*, in our model represented by class *Labeled*, is of great importance to *OShell*. In contrast to graphical object-oriented systems that allow to simply point at some visible representation of an object, a command-line oriented system must rely on names to reference objects. The object names map - in general - to filenames of the underlying filesystem. *OShell* has some more basic concepts in common with his file-oriented cousin.

An OShell process *navigates* through the object base's namespace (analogous to a file-shell navigating through the filesystem) thereby altering part of its *context*. The context of OShell roughly corresponds to the Shell's environment, but is represented as an internal, non-persistent object. The context object represents the entire internal state of an OShell process.

While the idea of a context object smells - frankly speaking - a bit of ideology, it comes in handy to get rid of some conceptual inconsistencies. There are, for example, a number of useful programs, such as *date(1)* that can't be meaningfully subsumed as some class' method, while it is straightforward to think of date and time as part of OShell's state. The context idea doesn't solve all our problems, though (e.g. *echo(1)*).

Accessing the objects

Objects aren't just the contents. Attributes are an essential feature of objects and capable to implement relationships to other objects. This makes it possible to access objects *implicitly*, i.e. without knowledge of their name or location. Let's assume, we have a code object *parse.y* which has an attribute "spec" of type *BnfGrammar* (subclass of *Specification*). Because attributes of class-types are realized as references to corresponding objects, the value must either be void or point to another object. Let's furthermore assume that a specification object exists, and has itself an attribute "requirement", of type *CustomerMemo*. The command

```
show parse.y!spec!requirement
```

indirectly accesses the requirements document through a code module that relates to its specification, and requirements objects. The exclamation mark is OShell's syntax to select an object's attributes. In the example above, OShell accesses the attribute "spec" of *parse.y*, retrieves the referenced *BnfGrammar* object, accesses its "requirement" attribute, and sends the referenced *CustomerMemo*, the message "show". A command-construction of this kind would, for example, allow a developer (or maintainer) to search for the idea behind an unclear design decision.

Although the notion of *versions* of design objects is separately modeled by class *History*, it is a built-in, access to versions is a built-in capability of OShell. It would have been awkward to reconcile access to versions with the outlined command syntax. Instead, we decided to support access to versions as an OShell language feature. To view for example a particular revision of *parse.y* one can simply type

```
show parse.y#1.9
```

Before a message is actually sent to an object, OShell retrieves its proper revision. The

complementary notion of *version-binding* allows to select particular revisions conveniently. Version-bindings can be numbers (numbering schemes may vary) or symbolic names (e.g. *last*, or *release4*).

Processing of commands

When OShell has parsed a command line into a message, a list of recipient objects, and a list of arguments, it proceeds as follows:

- compose the *annotated message* from the message and the arguments
- for each object in the recipient list, read the attribute defining the object's class (also considering the version of the class)
- for each referred class, look up the class object (from a well known location, e.g. a local library or a name service; class objects are cached in OShell's virtual memory, once they have been looked up)
- for each recipient object, match the annotated message against the *message patterns* associated with the *methods* of its class
- in case a method corresponding to the message was found, execute the method with the object reference as first and the arguments as remaining parameters

Pipes and redirections

The CHief definition language is used to describe the external behavior of tool processes. Part of this behavior is sending a certain kind of data to standard-out, or expecting a certain kind of data from the standard-in channel. The type of data that is associated with a process' I/O channels is defined by the types associated with a function's plugs in the CHief specification. In order to be able to construct typesafe pipes, OShell and CHief share the notion of transient objects, existing only for an instance of time, when they are either produced (sent to stdout) or consumed (gobbled from stdin). When analyzing a pipe-commandline, OShell is able to tell whether the connection of method processes is type-safe. Example:

```
show parse.y#1.9 | grep , %token
```

will cause OShell to retrieve revision 1.9 of *parse.y*, and send it the message *show*. For simplicity's sake, let's assume that *show* is part of class *YaccSource*'s protocol, and defined like:

```
show () {
    "print contents of object."
    Effects: None
    Plugs: YaccSource Out;
}
```

The call will result in the creation of a transient object of type `YaccSource`. Because the `grep` message does not have a *primary receiver* (i.e. an object reference to its right), it is sent to the transient object (*secondary receiver*), "coming in" from the left side. If we assume `YaccSource` to be a descendant class of `Text` (providing `grep`), a corresponding function can be found. `Grep`'s specification (see example above) declares the `In` and `Out` plugs to be of type `Text`. The function is found to be applicable, because we can safely connect the `Out`-plug of `show` with `grep`'s `In`-plug⁸. As the command line doesn't say anything about the `Out`-object of `grep`, it will rush across the screen and cease to exist. It is - of course - possible to catch temporary objects:

```
show parse.y#1.9 | \
  grep , %token > toks
```

In this example, `OShell` would create a new object `toks` of type `Text`.

Problems and Perspectives

It is our understanding, that the platform on which future SDEs will be built is the computer network. `OShell` and `AtFS` have both been inspired by this idea. `AtFS` has been completely redesigned, with networkability as one major criteria, so objects can be shared or accessed within local- and wide-area networks. The concept of a remote shell is already well known. However, we want to go a step further in network support for `OShell`: it shall provide a network-wide object service, usable by alternate front-ends or other tools. The idea of eventually having a nice graphical user interface (UI) "sitting on top" of the command interpreter, also consistently influenced the design of the described system.

Although specific problems for UI support and networking are different, the abstract service interface is important for both. We are now in the process of designing an alternate command interface for `OShell`, the *Annotated Message Protocol* (AMP), providing a generalized access service for the object base. The basic concept of `OShell` as AMP server is taking *requests*, and providing *feedback* in response to requests. Requests have the form:

```
<annotated msg, recipients>
```

Recipients are object handles, annotated messages are composed of the message name and parameters. Providing appropriate feedback is a much more complex problem. The `OShell` protocol shall provide for various requirements on behalf of possible clients. To come to a better understanding of these problems, we are implementing a graphical front-end that provides a desktop-like UI. One of the problems is

⁸To remain consistent with the metaphor, one of the connecting ends should be a *socket* - but that's a different story.

to keep the state of the displayed objects consistent with the objects in the object base. Status changes may happen asynchronously. How much state shall be possibly displayed - it does certainly make sense to have more states than *selected*, and *deselected* for software objects. Also, the objects that are to be displayed must be reported to the UI client on startup of a session (or a context change).

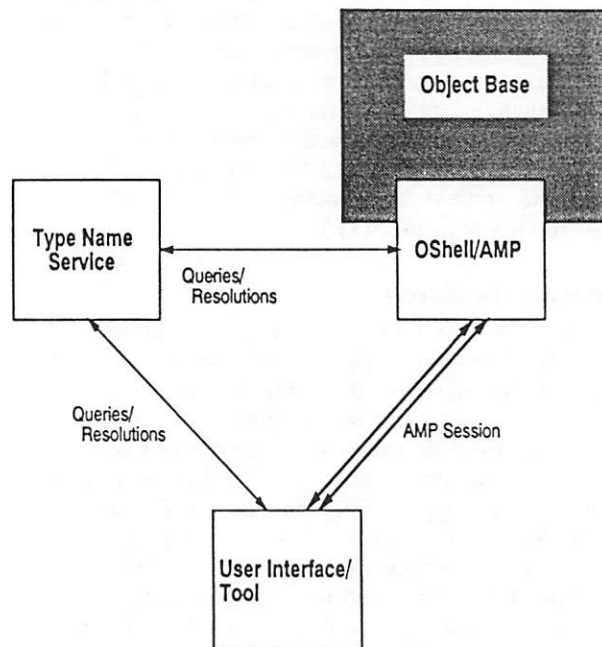


Figure 3: AMP service and client

Another problem is to control method-execution characteristics. There will be methods that have no visible effects, but there will also be interactive methods that need a terminal emulator. Other methods will open their own window on the workstation display. `CHieF` does offer some support here by allowing to specify method-characteristics, such as *Window*, *Tty*, or *Invisible*. There won't be any provisions for keeping the overall appearance of a graphical UI consistent, as it is done in `FIELD` or `Softbench`. Instead, we rely on UI style standards, such as *Motif* or *OpenLook* that will eventually be universally accepted. For an `OShell/AMP` client, there will only be the conceptual requirements, that a mapping of object-representatives in the client's virtual machine to objects in the object base be maintained, and that the *context* of the `OShell` process is understood.

Some other problems

One of the described system's strengths - but also vulnerabilities - is that its objects are files. As long as these are created and altered in the object oriented discipline, the system has a chance to keep track of the objects' class information and attributes. However, there are also zillions of files that haven't

been created this way. As a matter of fact, when we install an object base, we want to be able to check already existing files into our type system. It is also necessary to handle objects that come in from "outside", for example from a tape. Our current approach is to use filename-suffixes as hints concerning a file's type. For all objects that can't be typed we have to assume a default class. A solution to this problem will be a *typing-tool* that would act as hyper-intelligent *file(1)* program. The program should be able to tell the class of an object by looking at its file properties and its contents. The characteristics-pattern that qualifies an object as being of a certain class should ideally be part of the class definition.

Implementation

Writing a completely new shell-like command interpreter is a rather ambitious enterprise for a small project group like our's. This holds in particular if the command-interpreter shall be suited for real world use. When we designed OShell, we were aware that it would be unrealistic to write it completely from scratch. What made our idea appear feasible was the prospect of taking an existing shell implementation and modify it according to our design. We decided to take FSF's shell implementation, the *Bourne Again Shell* (BaSh) as starting point for OShell. All the nice things everyone wants to have in a shell, like command history, commandline- and history editing, job control etc., are already there and need only little or no modification at all. The filename completion feature is particularly useful and extended to attribute completion. To change the syntactical front-end of the program is very straightforward.

The object base was also no problem because we already had one, the aforementioned AtFS. As a matter of fact, AtFS and the version control, and configuration management system are also free software and publically available from a number of sites.

A first implementation of the object system is based on AtFS 1.1, an early version of CHieF and BaSh 1.05. It provided many insights that have led to the improved concept and design described in this paper. However, to realize these concepts required a redesign of several key components. Namely CHieF in its first version turned out to be insufficient and had to be redesigned, almost from scratch. We expect to have a reasonably operational system by summer.

Conclusion

Using an object-oriented command interpreter as user interface to an enhanced UNIX system (*enhanced*, because of the notion of *software object*) implies a more organized approach to using the UNIX environment. Rather than using the raw power

of (some) tools that happen to be available, one (actually some sort of *environment maintainer*) has to *organize* the tools around classes of software objects with well defined properties. While requiring the investment of some extra analytic thinking when setting up an environment, this approach adds a great deal of meaning to the objects that once were "stupid" containers for bytes, called *files*.

Adding support for a new kind of software object basically consists of defining the respective class, possibly using predefined functionality (such as version control) via the inheritance mechanism, and implementing the corresponding methods in form of calls to UNIX tools (e.g. version control programs), or (O)Shell scripts that are also stored in the object base as special software objects of class Method. By applying this idea, we are able to amalgamate the prototyping power of UNIX-typical tool combination (shell scripts, pipelines) with a formally integrated behavior of the environment.

While OShell itself is not a graphical user interface, it can be used to interact with a graphical environment *shell-tool*. In fact, the object shell represents an *abstraction* that has a well defined syntactical interface, suitable for formally communicating with a user interface process that presents the concepts of the object shell in a graphical way on a workstation.

With its loosely coupled tool integration concept and the rather simple way of dealing with object-base schema data, OShell is easily extensible for use in distributed environments. The flexibility that is known from using remote shells will directly be obtainable from a concept of remote OShell. The specification of an abstract object base service interface (AMP) offers even more potential for networked object applications.

Acknowledgement

This work is part of the STONE project. STONE - a STructured and OpEn Environment is a research project supported by the German Ministry of Research and Technology (BMFT) under grant ITS-8902E8. It is a joint project of nine institutions in the research and education domain of (former) East- and West-Germany. Participants from West-Germany are five research institutes and one university. Participants from East-Germany are one research institute and two universities.

References

1. Axel Mahler and Andreas Lampen, "An Integrated Toolset for Engineering Software Configurations," *Software Engineering Notes*, Vol. 13, No. 5, pp. 191-200 ACM Press, (November 1988).
2. Jeffrey C. Mogul, *Representing Information about Files*, PhD Thesis, Department of

- Computer Science, Stanford University, Report No. STAN-CS-86-1103, Stanford, California, USA (March 1986).
3. Leon Osterweil, "A Process Object Centered View of Software Environment Architecture," *International Workshop on Advanced Programming Environments*, pp. 156-174 Springer Verlag, (June 1986).
 4. Geoffrey M. Clemm, "The Workshop System - A Practical Knowledge-Based Software Environment," *Software Engineering Notes*, Vol. 13, No. 5, pp. 55-64 ACM Press, (November 1988).
 5. Richard N. Taylor, Richard W. Selby, Michael Young, Frank C. Belz, Lori A. Clarke, Jack C. Wileden, Leon Osterweil, and Alex L. Wolf, "Foundations for the Arcadia Environment Architecture," *Software Engineering Notes*, Vol. 13, No. 5, pp. 1-13 ACM Press, (November 1988).
 6. Timothy A. Budd, "The Design of an Object Oriented Command Interpreter," *Software - Practice and Experience* 19,1 pp. 35-51 (January 1989).
 7. Walter F. Tichy, "Tools for Software Configuration Management," *Proceedings of the International Workshop on Software Version and Configuration Control*, pp. 1-20 German Chapter of the ACM, (January 1988).
 8. Geoffrey M. Clemm, "The Odin System: An Object Manager for Extensible Software Environments," *CU-CS-314-86*, The University of Colorado, (February 1986).
 9. William Courington, "The Network Software Environment," *A Sun Technical Report*, Sun Microsystems, Inc., (May 1989).
 10. PCTE, *PCTE - A Basis for a Portable Common Tool Environment*, Commission of the European Communities, Brussels, Belgium (November 1988). Functional Specification, Version 1.5
 11. Steven P. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software* 7(4) pp. 57-66 IEEE Computer Society, (July 1990).
 12. Martin R. Cagan, "The HP SoftBench Environment Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal* 41(3) pp. 36-47 Hewlett-Packard Inc., (June 1990).
 13. Brian D. Fromme, "HP Encapsulator: Bridging the Generation Gap," *Hewlett-Packard Journal* 41(3) pp. 59-68 Hewlett-Packard Inc., (June 1990).
 14. Christer Fernström and Lennart Ohlsson, "ESF — An Approach to Industrial Software Production," pp. 17-28 in *Keith H. Bennet (Ed.)*, John Wiley & Sons (1989).
 15. Axel Mahler and Andreas Lampen, "Integrating Configuration Management into a Generic Environment," *Symposium on Practical Software Development Environments*, ACM Press, (December 1990).
 16. Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall International (1988).
 17. Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley (1986).
 18. Stanley B. Zdonik, "Version Management in an Object-Oriented Database," *International Workshop on Advanced Programming Environments*, pp. 405-422 Springer Verlag, (June 1986).

Axel Mahler is researcher at the Technische Universität Berlin. He has been working on several research projects on software engineering environments and software configuration management. His research interests are cooperative environments, object oriented systems, and networks.



Axel received his Diplom (MS degree) in computer science from the Technische Universität Berlin in 1985. His electronic mail address is axel@coma.cs.tu-berlin.de. Surface mail should be directed to TU Berlin, Sekr. FR 5-6, Franklinstr. 28/29, D-1000 Berlin 10, Germany.

The Process File System and Process Model in UNIX System V

Roger Faulkner – Sun Microsystems
Ron Gomes – AT&T Bell Laboratories

ABSTRACT

We describe the process file system `/proc` in UNIX System V Release 4 and its relationship to the UNIX process model abstraction. `/proc` began as a debugger interface superseding `ptrace(2)` but has evolved into a general interface to the process model. It provides detailed process information and control mechanisms that are independent of operating system implementation details and portable to a large class of real architectures. Control is thorough. Processes can be stopped and started on demand and can be instructed to stop on events of interest: specific machine faults, specific signals, and entry to or exit from specific system calls. Complete encapsulation of a process's execution environment is possible, as well as non-intrusive inspection. Breakpoint debugging is relieved from the ambiguities of signals. Security provisions are complete and non-destructive.

The addition of multi-threading to the process model motivates a proposal for a substantial change to the `/proc` interface that would replace the single-level flat structure with a hierarchy of directories containing status and control files. This restructuring would eliminate all `ioctl(2)` operations in favor of `read(2)` and `write(2)` operations, which generalize more easily to networks.

Introduction

The process file system represents all processes in the system as files in a directory conventionally named `/proc`. This concept was first introduced by Tom Killian in the research Eighth Edition UNIX system [1]. In System V Release 4 (SVR4) the concept has been refined from a simple replacement for the `ptrace(2)` system call into a general interface to the UNIX process model abstraction.

A typical "`ls -l /proc`" is shown in Figure 1. The name of each entry is a decimal number corresponding to the process id. The owner and group of the file are the process's real user-id and group-id, but permission to open the file is more restrictive than traditional file system permissions. The reported "size" is the total virtual memory size of the process; system processes such as process 0 and process 2 have no user-level address space, so their sizes are zero.

Standard system call interfaces are used to access `/proc` files: `open(2)`, `close(2)`, `lseek(2)`, `read(2)`, `write(2)`, and `ioctl(2)`. Data may be transferred from or to any valid locations in the process's address space by applying `lseek` to position the file at the virtual address of interest followed by `read` or `write`.

A process file contains data only at file offsets that match valid virtual addresses in the process. I/O operations with a file offset in an unmapped area fail. I/O operations that extend into unmapped areas do not fail but are truncated at the boundary. This includes writes as well as reads.

Information and control operations are provided through `ioctl`. A few of the `ioctl` operations are:

<code>PIOCSTATUS</code>	Get process status.
<code>PIOCSTOP</code>	Direct process to stop and ...
<code>PIOCWSTOP</code>	Wait for process to stop.
<code>PIOCRUN</code>	Make stopped process runnable.

-rw-----	1	root	root	0	Oct	31	10:06	00000
-rw-----	1	root	root	208896	Oct	31	10:06	00001
-rw-----	1	root	root	0	Oct	31	10:06	00002
...								
-rw-----	1	rrg	staff	131072	Oct	31	10:06	00206
-rw-----	1	weath	staff	749568	Oct	31	10:06	00370
-rw-----	1	raf	staff	651264	Oct	31	10:06	00393

Figure 1: A sample `/proc` directory

PIOCTRACE	Define set of traced signals.
PIOCSFAULT	Define set of traced machine faults.
PIOCSENTRY	Define set of traced syscall entries.
PIOCSEXIT	Define set of traced syscall exits.
PIOCGREG	Get values of process registers.
PIOCSREG	Set values of process registers.
PIOCMAP	Get virtual address mappings.

This list is not exhaustive. Some of these operations are explained in more detail and additional ones are introduced in the following sections. Others are omitted entirely for brevity. The SVR4 *proc(4)* manual page provides complete details.

Process Address Space

SVR4 incorporates a new Virtual Memory (VM) architecture (derived from SunOS) that provides processes much greater control over the structure and content of address spaces [2, 3]. A process executes in a virtual address space consisting of a number of *memory mappings* (contiguous virtual address ranges). Associated with each mapping are a virtual address, a length, and a set of flags describing permissions (read, write, execute) and other attributes.¹ The traditional notions of text, data, and stack do not appear explicitly in this model but are subsumed by more general notions.

New system calls permit a process to map objects (generally files) into and out of its address space (*mmap(2)*, *munmap(2)*) or to change the protections on a mapping (*mprotect(2)*). A mapping can be *private* (*MAP_PRIVATE*) or *shared* (*MAP_SHARED*). Modifications to a shared mapping are reflected through to the mapped object and appear in the address space of all other processes with a shared mapping to that object. Modifications to a private mapping affect only the address space of the process making the change and are invisible outside that address space.

The fact that a mapping is “private” does not mean that the implementation prohibits memory-sharing among processes that are mapping the same object. In fact, private mappings are implemented so as to provide *copy-on-write* semantics. Multiple private mappings to an object share the same memory pages until a process attempts to modify such a shared page, at which time the page is copied and the copy replaces the original in the address space.

¹The granularity of a mapping is a system-specific page size, typically a small multiple of 1024 bytes. There is more to the VM architecture than is presented here. For example, individual pages can be mapped with different permissions and to different underlying objects.

Within this model a “text” segment is nothing more than a private executable mapping to the code portion of an executable file, *i.e.* an *a.out*. A “data” segment is a readable and writable private mapping to that portion of an *a.out* containing initialized data. A “stack” segment is a read/write mapping into which the stack pointer points, but is otherwise undistinguished (and in fact a sophisticated application can have multiple stacks). The system provides suitably-behaving anonymous objects to which mappings may be applied in the construction of other segments (*e.g.* “bss”, uninitialized zero-filled memory). Shared libraries are implemented by mapping the code and data of a shared library executable file into the address space of a process.

The *PIOCMAP* operation extracts the memory map of a process. Figure 2 shows a typical memory map, obtained by a simple tool that reports the contents of the map structures returned by *PIOCMAP*. The list contains a number of writable mappings (presumably data) and a number of mappings that are read-only and executable (presumably code), from both the *a.out* itself and a shared library that has been mapped.²

80000000	26K	read/exec
80008000	6K	read/write/exec
80009800	74K	read/write/exec/break
C0020000	4K	read/write/exec/stack
C1000000	148K	read/exec
C1026000	4K	read/write/exec
C1027000	2K	read/write/exec
C1028000	2K	read/write

Figure 2: A Typical Memory Map

What may not be apparent from this list is that all the mappings are private (this is generally the case unless processes explicitly arrange to communicate with one another through a shared mapping). In particular the code portions are *MAP_PRIVATE* mappings with read and execute permissions. What happens if an attempt is made to store into a code portion? The process itself can’t do this directly (reasonably so) because it doesn’t have write permission on the mapping, but a controlling process can write the address space through the */proc* interface. In this case the system will permit the write, and copy-on-write semantics will be provided where necessary. In this way breakpoints can be planted in

²Note that “stack” and “break” mappings appear in the list despite all the disclaimers. The operating system is prepared to grow one mapping (the initial program stack segment) automatically and another (the break segment) on explicit request by the *brk(2)* system call. A process-control application can sometimes make use of this information so it is provided in the *PIOCMAP* interface.

code, or data modified, without corrupting either the `a.out` file being executed or the address space of other processes that may be executing the same code.

Process Context

The execution context of a process (at least the portion deemed relevant) is described by the `prstatus_t` structure, which a controlling process can request at any time. Elements of this structure describe signal state, the contents of processor registers, process and session ids, and scheduling state (running or stopped, with more detailed information about stopped processes). The structure is returned by the `PIOCSTATUS` request or as an optional side-effect of the process-stop requests `PIOCSTOP` and `PIOCWSTOP`; it is designed to contain the information most frequently needed by a controlling process such as a debugger. Other data structures and operations exist for details of process state that are less frequently used, such as the contents of the floating-point registers, the information needed by `ps(1)`, and the signal actions for every signal. Process state can be modified in controlled ways; for many of the operations that "get" state information there is a corresponding "set" operation. Thus, for example, the floating-point registers are fetched into a structure of type `fpregset_t` by the `PIOCGFPREG` request and are modified by the `PIOCSFPREG` request.

An important difference between this style of interface and that provided by the research prototype is the presentation of a complete and consistent *process model* as independent as possible of internal system implementation details. Formerly it was necessary to examine and directly manipulate the *user* and *proc* structures of the target process in order to effect state changes; this tied a process-control program to details that could (and did) change between releases of the system and was a functional improvement over *ptrace* only to the extent that it provided greater bandwidth and the ability to control unrelated processes. A primary goal was to remove these dependencies from the interface; secondary goals were to ease debugger development, improve portability of applications, and reduce the number of system calls routinely made by a debugger.³ This has an associated cost in that there are more operations and data types for the programmer to master, but the cost is small in comparison with the resultant improvements in capability, consistency, portability, and efficiency.

³The goal of debugger efficiency, though irrelevant in many situations, becomes important in the implementation of features such as conditional breakpoints, for which "breakpoints per second" is a realistic measure of performance.

Events of Interest

A process executes in an environment established by and enforced by the UNIX kernel. Natural points of control for a process are where it enters and leaves the kernel, specifically, system call entry and exit, machine faults, and receipt of signals.

Events of interest are specified through the `/proc` interface using sets of flags. Signals are specified using the POSIX signal set type, `sigset_t`. Machine faults and system calls are specified using analogous set types `fltset_t` and `sysset_t`. Like signals, faults and system calls are enumerated from 1; there is no fault number 0 or system call number 0.⁴ The SVR4 implementation provides for up to 128 signals, 128 faults and 512 system calls.

A traced process stops when it encounters an event of interest or when it is directed to stop, normally because the controlling process issued a `PIOCSTOP` request. It may also stop for reasons external to `/proc`; the competing mechanisms for stopping a process are *ptrace* and job-control stop signals.⁵ (The `/proc` stop directive is independent of signals.) Ignoring the competing mechanisms, points in the kernel at which a process may stop are illustrated in Figure 3.

A stop on system call entry occurs before the system has fetched the system call arguments from the process. A stop on system call exit occurs after the system has stored all return values in the traced process's data and saved registers. This gives a debugger the opportunity to change the system call arguments before processing occurs and to manufacture whatever return values it wishes the process to see. In addition, a process that is stopped on system call entry can be directed to abort execution of the system call and go directly to system call exit. This combination of facilities enables complete encapsulation of the system call execution environment of a process so that, for example, older system calls or alternate versions of them can be simulated entirely at user level. (This is one way in which obsolete facilities could be supported "forever" without cluttering up the operating system.)

Stopping on machine faults and on system call entry and exit is straightforward; the process simply enters the kernel and stops. Stopping on receipt of a signal is more involved.

There are basically two points in the kernel where signals are detected: when the process is returning to user level and when the process is sleeping at an interruptible priority within a system call.

⁴System call number 0 exists in some UNIX system implementations as the "indirect" system call, but this only provides an alternate method for passing the real system call number.

⁵*ptrace* is made obsolete by `/proc` but is still required by the System V Interface Definition.

The kernel function *issig()* handles both cases.

Just before a process returns to user level, it checks for the presence of a signal to be acted upon and then acts on it by executing:

```
if ( issig() )
    psig();
```

If there are non-held and non-ignored signals pending for the process, *issig()* promotes one of them from pending to current and returns true. If the action for the signal is SIG_DFL, *psig()* terminates the process, possibly with a core dump. Otherwise, *psig()* modifies the saved registers and the user-level stack so that the process will enter the signal handler for the current signal when execution is resumed at user level. Job-control stop signals are treated differently; the default action for these signals is taken within *issig()*.

Within an interruptible sleep, *issig()* is called to determine if the system call should be terminated with EINTR. If so, the process returns to *syscall()*, perhaps stopping on *syscall* exit along the way, to ask the question again. Since there is already a current signal, another signal is not promoted by the second call to *issig()*.⁶

issig() handles all cases of stopping the process due to receipt of a signal as well as the case of stopping the process due to the presence of a */proc* stop directive. This includes stopping the process by the competing mechanisms. The complete logic of *issig()* is illustrated in Figure 4.

A process may stop twice due to receipt of a job-control stop signal, first on a signalled stop if the signal is being traced and again on a job-control stop if the process is set running without clearing the signal. A job-control stop is not an event of interest to */proc*. Such a stopped process can be restarted only by sending it a SIGCONT signal. However the process can be directed to stop via */proc* so that, when restarted by SIGCONT, it stops again on a requested stop before exiting *issig()*. */proc* gets the last word.

A similar situation holds for *ptrace*. When controlled via *ptrace*, a process stops on receipt of any signal, whether or not that signal is included in the set of signals traced via */proc*. If the signal is traced via */proc*, the process must be set running through */proc* before it can be manipulated by *ptrace*. Even though the process is logically set running, it remains stopped on the signalled stop and cannot be set running again through */proc*; *ptrace* has control. After *ptrace* sets the process running, it will stop again on requested stop before exiting *issig()* if it was directed to stop through */proc*.

⁶Older UNIX systems did not use the current signal concept and consequently suffered a race condition in which the signal detected by *issig()* might not be the signal actually delivered to the process by *psig()*. This caused a variety of problems, including a possible panic of the operating system if *psig()* attempted to deliver an ignored signal. For debuggers the consequence was that all signals except perhaps one had to be cleared on restarting a process after a stop, not just the signal that caused the stop.

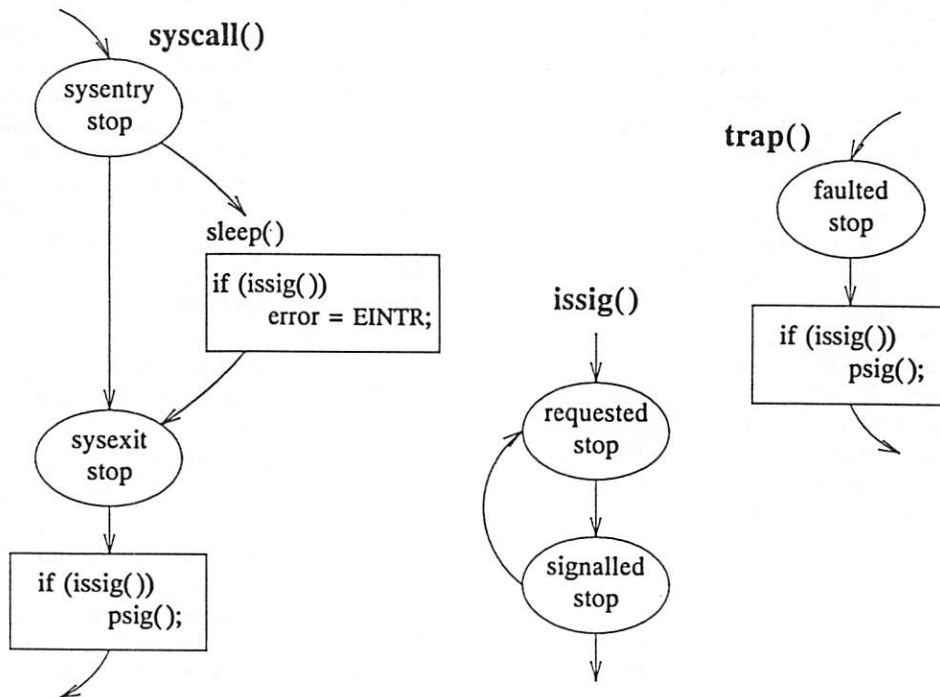


Figure 3: Events of Interest

/proc gets the first and last words.

Sending SIGCONT to a stopped process sets it running only if it is in a job-control stop; neither *ptrace* nor /proc can restart a job-control stopped process. All three mechanisms peacefully coexist by virtue of the delicate balance maintained in *issig()*, with cooperation from *setrun()*.⁷

An important consequence of having all signal-related stopping confined to *issig()* is that a signal received while asleep in an interruptible system call need not cause premature exit from the system call when the process is set running again.⁸ (The current signal can be cleared by the debugger. It is automatically cleared on a job-control stop; the SIGCONT signal that restarts the process will be discarded unless it is being caught.) Since the reason for sleeping may have gone away, *sleep()* must return normally to its caller when a stopped process is restarted without a current signal. Non-interruption of sleeping system calls relies on all callers of *sleep()* to test the reason for sleeping and

to call again if the condition is still true, typically:

```
while ( condition )
    sleep(...);
```

This is a fine point and a fruitful source of kernel bugs.

Because a requested stop is performed in *issig()*, a process can be directed to stop while it is sleeping and set running again without disturbing the system call. The process can also be directed to abort the system call without having to send it a signal.

Breakpoints

The /proc interface does not directly implement the concept of a process breakpoint, but it provides sufficient mechanism for a debugger to do so. Breakpoints can be installed in a process by a debugger using the *read* and *write* operations on the process address space to replace the machine instruction at each breakpoint address with an illegal user-level instruction. Most systems designate one instruction as the approved "breakpoint" instruction,

⁷The *ptrace* and job-control stop mechanisms have always been in conflict. Job-control stops used to be disabled when a process was controlled by *ptrace*.

⁸UNIX systems used to arrange for signalled stops to occur within *psig()*, thereby forcing EINTR failures of interruptible system calls.

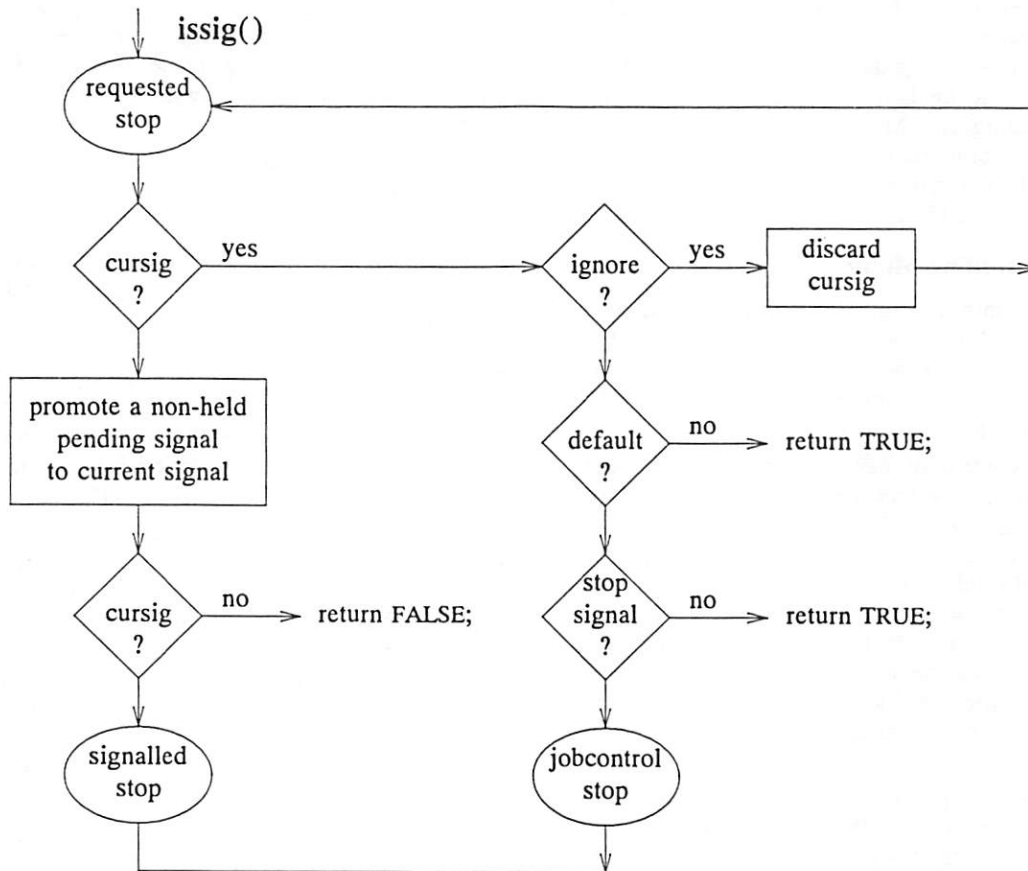


Figure 4: Process Control in *issig()*

but all that is really needed is one that causes a trap to the kernel. On architectures with variable-length instructions, the length of the breakpoint instruction should be that of the shortest instruction in the instruction set (to avoid overwriting the instruction following the breakpoint). The execution of the breakpoint instruction should leave the program counter with a known value relative to the breakpoint address in all cases, preferably the breakpoint address itself.

When the controlled process executes a breakpoint instruction, it takes a machine fault, FLTBPT if the instruction is the approved breakpoint instruction, otherwise FLTILL or FLTPRIV for a general illegal or privileged instruction. The process will stop on a faulted stop if the debugger has specified the particular fault as an event of interest. Otherwise the process is sent a signal, normally SIGTRAP or SIGILL. If the signal is not being held (blocked) by the process, the process will stop on a signalled stop if the debugger has specified receipt of the particular signal as an event of interest. The essential difference between stop-on-fault and stop-on-signal is the phrase, "if the signal is not being held."

A signal does not cause a process to stop when it is generated, only when it is received by the process. Also, any signal can be sent to a process by another process (subject to permissions). Lastly, there can be more than one signal pending for a process at one time. Signals are too overloaded in semantics and mechanism to be used reliably for breakpoint debugging. Machine faults are not used for inter-process communication and cannot be intercepted or held by a process; stop-on-fault is the preferred method for fielding breakpoints.

Controlling Multiple Processes

When a controlled process creates a child process, the controlling process may wish to add the new process to its set of controlled processes or it may wish to let the new process run unmolested. In either case some action must be taken.

To take control of new processes, a debugger can set the *inherit-on-fork* flag in the original process and arrange to trace exit from the *fork(2)* and *vfork(2)* system calls. When the controlled process forks, the child inherits all of the parent's tracing flags and both parent and child stop on exit from the *fork*. The debugger sees the parent's stop on exit from *fork* and uses the return value (the pid of the child) to open the child's */proc* file. Because the child stopped before executing any user-level code, the debugger can maintain complete control.

To allow new processes to run unmolested, the debugger can simply reset the *inherit-on-fork* flag so that new processes start with all tracing flags cleared. However, if breakpoints have been set any new process will inherit them and possibly

malfunction. In this case the debugger must arrange for the controlled process to stop on entry to as well as exit from *fork* and *vfork*. When the controlled process stops on entry to *fork*, the debugger lifts all the breakpoints and sets the process running. The child starts running with no tracing flags and no breakpoints. The parent stops on exit from *fork* and the debugger can replant all the breakpoints. Special care must be taken with *vfork* because the address space is shared between parent and child until the child *exits* or *execs*. */proc* provides sufficient mechanism to deal with this case efficiently.

Miscellaneous

Tracing flags can remain active for a process when its process file is closed, allowing a process to be left hanging and later reattached by a debugger. This behavior is changed by setting the *run-on-last-close* flag. When this flag is set and the last writable */proc* file descriptor for the process is closed, all of the tracing flags are cleared and, if the process is stopped, it is set running. This can be used by a controlling process to ensure that its controlled processes are released even if it itself is killed with SIGKILL.

Given a virtual address in the controlled process, the *PIOCOPENM* operation returns a read-only file descriptor for the underlying mapped object, if any. This enables a debugger to find executable file symbol tables, including those for shared libraries attached to the process, without having to know pathnames.

The *PIOCCRED* and *PIOCGROUPS* operations return complete credentials information for the controlled process.

Finally, the *PIOCGETPR* and *PIOCGETU* operations return, respectively, the *proc* structure and *user* area for the controlled process. These operations are provided for completeness but their use is deprecated because a program making use of them is tied to a particular version of the operating system. Their very existence reveals details of system implementation and their continuation into the new world of multi-threaded processes is doubtful.

A number of things that might be useful to know about a process are not provided through the */proc* interface, such as its file creation mask. Our approach has been to provide information and control operations for the most common things that a debugger needs, and for things that a process cannot discover or do to itself through system calls. For the remainder, a debugger can force a process to execute system calls on the debugger's behalf without the process's knowledge or consent.

It is worth noting that the SVR4 implementation of */proc* works correctly with Remote File Sharing (RFS) [4]. With appropriate permission it is possible to inspect, modify and control processes running on

any machine in an RFS network. This extension of capability "for free" to any machine in the network applies to any resource that is accessible within the file system name space and is an additional justification for implementing resources this way.⁹

Integrity and Security

The interface distinguishes operations that modify process state or behavior (such as a request to write the registers) from those that merely inspect process state (such as a request for process status). The former are regarded as "read/write" operations and the latter as "read-only." A `/proc` file can be opened for exclusive read/write use (if `O_EXCL` is specified in the `open(2)`); in this way a controlling process can avoid collisions with other controlling processes. Read-only opens are unaffected in this case.

All I/O and control operations are guaranteed to be atomic with respect to the traced process. Copy-on-write is performed by the system excepting only bona-fide shared memory; writing to one process will not corrupt another process executing the same executable file or shared library. This applies in general to `MAP_PRIVATE` VM mappings.

Permission to open a `/proc` file requires that both the uid and gid of the traced process match those of the controlling process; `setuid` and `setgid` processes can be opened only by the super-user. When a traced process *execs* a `setuid` or `setgid` executable file, the set-id operation is honored but the file descriptor held by the controlling process becomes invalid; no further operation on that file descriptor will succeed except `close(2)`, thus enforcing security without modifying process behavior.¹⁰ When the set-id *exec* occurs, the traced process is directed to stop and its run-on-last-close flag is set. A controlling process with appropriate privilege can reopen the named `/proc` file to retain control of the target; just closing the invalid file descriptor clears all tracing flags and sets the set-id process running.

Implementation

The implementation of `/proc` as a set of "files" is facilitated by the Virtual File System (VFS) architecture of SVR4 which is derived from the *vnode* feature [5] of SunOS and subsumes the File System Switch (FSS) of earlier releases of System V. VFS permits the coexistence on a single system of several disparate *file system types* (fstypes) by providing a clean separation of file system code into *generic* (file system-independent) and *specific* (file

system-dependent) pieces with a well-defined but narrow interface between the pieces. (Generic code is viewed as "upper-level" and specific code as "lower-level.") Typically the set of fstypes on a system will include conventional disk file systems and network file systems as well as more outlandish things such as `/proc`. In general any resource can be made to appear within the file system name space if it makes sense to view it that way.

The fundamental data structure manipulated by generic code is the *vnode* (virtual node), which is the system's internal representation of a file and provides the handle by which file manipulations are performed. A *vnode* contains both public and private data. The public data in a *vnode* consists of information that is maintained by the upper level or that does not change over the life of the file (such as the file type); private data is opaque to the upper level and is implementation-specific (such as a list of block addresses for a disk file).

The upper level requests the creation of *vnodes* by the lower level, and these *vnodes* are subsequently supplied as operands to other file operations. The set of *vnode* operations includes *open*, *close*, *read*, *write*, *ioctl*, *lookup*, *create*, *remove*, and many more. The developer of a file system type provides the code that implements the necessary set of *vnode* operations for that type.

Within this framework the construction of the fantasy world (the illusion that processes are actually files) is straightforward. System call references to `/proc` files result in the invocation of lower-level code to create and maintain `/proc` *vnodes*. For example, an attempt to open `/proc/2846` results in a call to *prlookup* which searches the system process structures for process id 2846 and (if such a process exists) constructs a *vnode* for it. The upper-level code associates this *vnode* with the open file descriptor, and subsequent applications of *read(2)*, *write(2)*, and *ioctl(2)* result in calls to *prread*, *prwrite*, and *prioctl* to perform the requested process I/O or control operation. Similarly, a command like *ls(1)* that wants to read the contents of the `/proc` directory will apply *readdir(3)* to it; this results in a call to *prreaddir* which examines the system process structures and satisfies the system call by constructing a set of directory entries naming all the processes in the system.

The intimate connection with process control requires some code in addition to the usual VFS plumbing; in this respect `/proc` is an unconventional file system and not an "add-on." Most of this code deals with the interaction between signals and process stopping and appears in *issig()* (discussed above in more detail). Minor changes were made in a few other places including the system-call handler (to stop the process on system call entry or exit), the user trap handler (to stop the process when it incurs a machine fault), the scheduler (to suspend a process

⁹Needless to say, a debugger that takes advantage of this facility must be prepared to deal with all of the problems inherent in heterogeneous networks.

¹⁰This differs from the more intrusive behavior with *ptrace*, in which set-id flags are ignored if the target performs an *exec(2)*.

undergoing `/proc` I/O), `exec(2)` (to invalidate `/proc` file descriptors to set-id programs), and `exit(2)` (to inform `/proc` of the death of a process).

Implementation of `/proc` I/O requires that one process be granted access to the address space of another. This was a troublesome problem in the original research prototype because the memory management code of the underlying system made it difficult for one process to incur a page fault on behalf of another. The new Virtual Memory architecture simplifies this problem. VM provides a model of memory management in which machine-dependent details are isolated in a separate layer.

In particular, each process has an associated *address space* ("as") data structure to which a set of standard operations may be applied. One such operation is `as_fault`, which performs page-fault processing for a specified range of addresses. Given this operation, all that is necessary for inter-process I/O is for the controlling process to apply `as_fault` to the address space of the target process, map the target pages into its own address space, and copy the data between the two addresses.

Overall, a high degree of portability was achieved in the implementation of `/proc`. The VM abstraction hides many of the details of memory management. Machine-specific `/proc` VFS code is confined to a single source file containing less than 10% of the total `/proc`-related code. Assuming a complete implementation of the VM primitives and of the generic porting base, porting should require only the specification of a few details such as the code for fetching register contents. (There is a presumption here that the process model accommodates all "interesting" machines.)

Applications

The SVR4 `ps(1)` command is implemented using `/proc`. Special provision was made for it in the interface; the `PIOCPINFO` operation returns everything that `ps` might want to display about a process. The logic of `ps` is to read the `/proc` directory, open each process file in turn, issue the `PIOCPINFO` request, close the file, and print the result if appropriate according to the `ps` options. Because `ps` runs with super-user privilege and the process files are opened read-only, the `opens` always succeed and no interference is created for controlling and controlled processes. Because all the information for a process is obtained in a single operation, each line of `ps` output is a true snapshot of the process, even though the complete listing is not a true snapshot of the whole system.

The interception of system calls with `/proc` is at the heart of `truss(1)`, a command that traces the execution of a process, producing a symbolic report of the system calls it executes, the faults it encounters and the signals it receives. `truss` can be

applied to running processes or used to start up commands to be traced, and will optionally follow the execution of child processes as well. Because it requires no symbol information and is applicable at any time to an arbitrary process (even `init`), it can often be used to find out what a misbehaving program is *really* doing even if source is unavailable and the executable file symbol information has been stripped. `truss` output can be startling.

`truss` is constrained by the security provisions of `/proc`, so that it can be applied only to ordinary (non-set-id) processes owned by the user. Moreover if the traced process `execs` a set-id program `truss` loses control; the process continues normally, with correct credentials, but no longer under control of `truss`. If `truss` is run by the super-user, all permissions are granted and any process and all its children can be traced. `truss` will not alter the behavior of a process other than by slowing it down. (Of course, just slowing it down can affect behavior if the process uses `alarm(2)` or other real-time mechanisms.)

The interface is clearly intended to facilitate a sophisticated debugger and has already supported the development of several prototypes. Such a debugger is planned for a future release of the system. In the meantime the use of `ptrace` is being phased out; the standard debuggers `sdb(1)` and `dbx(1)` have been rewritten in SVR4 to use `/proc` (and, for `sdb`, to add a few new capabilities, such as the ability to grab and debug an existing process).

Proposed Extensions

A number of new facilities have been proposed for inclusion in future releases of the system. We describe a few of them here (though note that there is no promise that any of these will actually be provided anytime soon).

By appropriately defining what it means for a `/proc` file to be "ready" it would be possible to permit `/proc` file descriptors to be used with the `poll(2)` system call. This would make it much easier for a debugger to wait for any one of a set of controlled processes to stop on an event of interest while also waiting for events such as keyboard input from the user. It would offer more flexibility for multi-process debugger implementations than the current method of waiting for only a single process to stop; this flexibility will be even more important when there can be multiple threads of control within a single process.

`/proc` currently gives short shrift to performance aspects of the process model. A resource usage interface has been proposed, along with an interface to a process's page data whereby a performance monitor can sample page-level referenced and modified information for a process on intervals at will.

A generalized data watchpoint facility has been proposed and designed, based on the VM system's ability to re-map read/write permissions on individual pages of a process's address space. It can be implemented on any architecture capable of running SVR4 and can take advantage of specialized hardware when available. The interface accepts specification of watched areas of any size, down to a single byte. The traced process stops only when a watchpoint really fires; the system takes care of the details of recovering from machine faults taken due to references to unwatched data that happens to fall in the same page as watched data.

It is possible, with the addition of a small but ugly wart on the `/proc` interface, to eliminate `ptrace` from the operating system and implement it as a library function built on `/proc`. The difficult part is not with `ptrace` itself, but rather with the requirement that a process stop via `ptrace` be reported to the parent via `wait(2)`.

The current implementation does not permit a debugger to directly map the address space of the traced process via `mmap(2)`; access is possible only through explicit `read` or `write` system calls. Permitting `mmap` would provide no new capability *per se* but would allow very high-speed inspection or modification of the target's address space. Such a facility is under consideration.

Proposed Restructuring

The evolution of the operating system toward a process model incorporating shared address spaces and multiple threads of control places some strain upon the interface in its current form. A new structure is under consideration that would change the `/proc` file system from a flat structure to a hierarchical one containing a number of sub-directories and additional status and control files. The programming interface changes from one in which `ioctl(2)` operations are applied to open file descriptors in order to effect process control and interrogate process state to one in which process state is interrogated by `read(2)` operations applied to appropriate read-only status files and process control is effected by structured messages written to write-only control files. (A structure similar in concept but different in detail appears in Plan 9 [6].)

The change in model has a number of advantages independent of multi-threading considerations. Removing the dependence on `ioctl` simplifies the implementation of `/proc` in a network environment. The unstructured nature of `ioctl` operations and the variability of operand sizes and I/O directions make it difficult to cleanly separate the client/server interactions; `read` and `write` don't share these problems. In addition the use of a control file to which structured messages are written makes it possible to combine several control operations in a single `write`

system call; this can improve the performance of some applications for which the number of system calls is a bottleneck.

Of more relevance for the process model is that a directory hierarchy is a natural structure in which to present the relationship between a process and the individual threads-of-control that share its address space. Thread-ids of sibling threads appear as sub-directories within a hierarchy that has the process-id at the top.

Outstanding Issues and Future Work

`/proc` completes a long-incomplete process-model/debugger interface. Unfortunately, a process model interface built into the kernel can of necessity deal only with kernel interfaces. The Application Binary Interface (ABI) was introduced in SVR4. The ABI is not a kernel interface but a user-level shared library interface, with the shared library being provided by the purveyor of the system.

With the advent of the ABI, programming interfaces move from the kernel level to the shared library level. This is especially true for multi-threaded applications in an environment in which user-level threads may be multiplexed onto a smaller set of kernel threads. A debugger that deals with the user-level threads model must have access points in the threads library of the same power as the system call interfaces that `/proc` provides for kernel-level threads. A generalized shared library interface control mechanism would benefit debugging of applications in general.

As always, debugging lags development.

Acknowledgements

Dave Weatherford has been a serious user of `/proc` and has given us consistently good advice over the years.

Jonathan Shapiro conceived of and did most of the design for the proposed generalized watchpoint facility.

Much useful advice was given to us by other developers both at Bell Labs and at Sun Microsystems during the development of SVR4. In particular we thank Bill Shannon.

We especially wish to thank the following people who gave us their support in bad times as well as good: Phyllis Eve Bregman, Sheila Brown-Klinger, Blaine Garst, Barbara Moo, and Marilyn Partel.

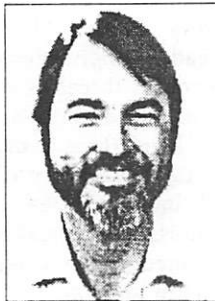
References

- [1] T. J. Killian, *Processes as Files*, Proceedings of the USENIX Association Summer Conference, Salt Lake City, June 1984, pp. 203-207.
- [2] R. A. Gingell, J. P. Moran, W. A. Shannon, *Virtual Memory Architecture in SunOS*,

Proceedings of the USENIX Association Summer Conference, Phoenix, June 1987, pp. 81-94.

- [3] Joseph P. Moran, *SunOS Virtual Memory Implementation*, Proceedings of the European UNIX User's Group, London, UK, April 1988, pp. 285-300.
- [4] A. P. Rifkin, M. P. Forbes, R. L. Hamilton, M. Sabrio, S. Shah, K. Yueh, *Remote File Sharing Architectural Overview*, Proceedings of the USENIX Association Summer Conference, Atlanta, June 1986, pp. 248-259.
- [5] S. R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, Proceedings of the USENIX Association Summer Conference, Atlanta, June 1986, pp. 238-247.
- [6] R. Pike, D. Presotto, K. Thompson, H. Trickey, *Plan 9 from Bell Labs*, Proceedings of the UKUUG Conference, London, UK, July 1990, pp. 1-9.

Roger Faulkner grew up in North Carolina. He received a B.Sc. in Physics from N. C. State University in 1963 and a Ph.D. in Physics from Princeton University in 1967, then joined Bell Laboratories. He was seduced by computers while using them to solve physics problems and joined the Murray Hill computer center in 1970.



During 1970-1975 he learned about the UNIX system by osmosis from denizens of the attic. During 1976-1980 he was actively involved in the inner workings of the UNIX kernel, after which he left Bell Labs to live and work in New York City. He returned to UNIX work in 1984 with a mission from God to create the one true debugger. He failed in this but succeeded in making Ron Gomes create the one true debugger interface. He hopes that others will use it for worthy purposes. His current work involves system support for debugging of multi-thread, multi-process, multi-machine, and multi-national applications.

Ron Gomes is a native Canadian, about which no more need be said. He received a B.Sc. in Mathematics from McGill University (Montreal) in 1975 and an M.Sc. in Computer Science from the University of Toronto in 1977. He has been working on and with UNIX systems since 1975 and knows what *dsw* means. Previous employers have included the University of Toronto, Bell-Northern Research, and HCR Corporation, for



whom he worked on projects including operating system support and enhancement, kernel ports, and compiler development. Since 1984 he has been with the System V development organization of AT&T Bell Laboratories, now UNIX System Laboratories, Inc., where he contributed to the design and development of System V Releases 3 and 4, notably in the area of file system architecture. His current work involves multi-processor systems and in particular the further development of the process model.

Limitations of the Kerberos Authentication System¹

Steven M. Bellovin, Michael Merritt – AT&T Bell Laboratories

ABSTRACT

The Kerberos authentication system, a part of MIT's Project Athena, has been adopted by other organizations. Despite Kerberos's many strengths, it has a number of limitations and some weaknesses. Some are due to specifics of the MIT environment; others represent deficiencies in the protocol design. We discuss a number of such problems, and present solutions to some of them. We also demonstrate how special-purpose cryptographic hardware may be needed in some cases.

INTRODUCTION

The Kerberos authentication system [Ste88, Mill87, Brya88] was introduced by MIT to meet the needs of Project Athena. It has since been adopted by a number of other organizations for their own purposes, and is being discussed as a possible standard. In our view, both these decisions may be premature. Kerberos has a number of limitations and weaknesses; a decision to adopt or reject it cannot properly be made without considering these issues. (A *limitation* is a feature that is not as general as one might like, while a *weakness* could be exploited by an attacker to defeat the authentication mechanism.) Some improvements can be made within the current design. Support for optional mechanisms would extend Kerberos's applicability to environments radically different from MIT.

These problems fall into several categories. Some stem from the Project Athena environment. Kerberos was designed for that environment; if the basic assumptions differ, the authentication system may need to be changed as well. Other problems are simply deficiencies in the protocol design. Some of these are corrected in the proposed Version 5 of Kerberos, [Kohl89] but not all. Even the solved problems merit discussion, since the code for Version 4 has been widely disseminated. Finally, some problems with Kerberos are not solvable without employing special-purpose hardware, no matter what the design of the protocol. We will consider each of these areas in turn.

We wish to stress that we are not suggesting that Kerberos is useless. Quite the contrary — an attacker capable of carrying out any of the attacks listed here could penetrate a typical network of UNIX systems far more easily. Adding Kerberos to a network will, under virtually all circumstances, significantly increase its security; our criticisms focus on the extent to which security is improved.

¹A version of this paper was published in the October, 1990 issue of *Computer Communications Review*.

Further, we recommend changes to the protocols that substantially increase security.

Beyond its specific utility in production, Kerberos serves a major function by focusing interest on practical solutions to the network authentication problem. The elegant protocol design and wide availability of the code has galvanized a wide audience. Far from a condemnation, our critique is intended to contribute to an understanding of Kerberos's properties and to influence its evolution into a tool of greater power and utility.

Several of the problems we point out are mentioned in the original Kerberos paper or elsewhere. [Davi90] For some of these, we present protocol improvements that solve, or at least ameliorate, the problem; for others, we place them squarely in the context of the intended Kerberos environment.

Version 5, Draft 3

Since this paper was written, a new draft of the Version 5 protocol has been released, and a final specification is promised. [Kohl90] Many of the problems we discuss herein have been corrected. Others remain, and we have found a few new ones. The ultimate resolution of these issues is unclear as we go to press. Consequently, a brief analysis of Draft 3 is presented in an appendix, rather than in the main body of the document.

Focus on Security

Kerberos is a security system; thus, though we address issues of functionality and efficiency, our primary emphasis is on the security of Kerberos in a general environment. This means that security-critical assumptions must be few in number and stated clearly. For the widest utility, the network must be considered as completely open. Specifically, the protocols should be secure even if the network is under the complete control of an

adversary.² This means that defeating the protocol should require the adversary to invert the encryption algorithm or to subvert a principal specifically assumed to be trustworthy. Only such a strong design goal can justify the expense of encryption. (No "steel doors in paper walls".) We believe that Kerberos can meet this ambitious goal with only minor modifications, retaining its essential character.

Some of our suggestions bear a performance penalty; others complicate the design of suggested enhancements. As more organizations make use of Kerberos, pressures to enhance or augment its functionality and efficiency will increase. Security has real costs, and the benefits are intangible. There must be a continuing and explicit emphasis on security as the overriding requirement.

Validation

It is not sufficient to design and implement a security system. Such systems, though apparently adequate when designed, may have serious flaws. Consequently, systems must be subjected to the strongest scrutiny possible. A consequence of this is that they must be designed and implemented in a manner that facilitates such scrutiny. Kerberos has a number of problems in this area as well.

WHAT'S A KERBEROS?

Before discussing specific problem areas, it is helpful to review Kerberos Version 4. Kerberos is an *authentication* system; it provides evidence of a *principal's* identity. A principal is generally either a user or a particular service on some machine. A principal consists of the three-tuple

$\langle \text{primaryname}, \text{instance}, \text{realm} \rangle$.

If the principal is a user — a genuine person — the *primary name* is the login identifier, and the *instance* is either null or represents particular attributes of the user, i.e., *root*. For a service, the service name is used as the primary name and the machine name is used as the instance, i.e., *rlogin.myhost*. The *realm* is used to distinguish among different authentication domains; thus, there need not be one giant — and universally trusted — Kerberos database serving an entire company.

Kerberos principals may obtain *tickets* for services from a special server known as the *ticket-granting server*, or *TGS*. A ticket contains assorted information identifying the principal, encrypted in

the private key of the service. (Notation is summarized in Table 1.)

$\{T_{c,s}\}K_s = \{s, c, \text{addr}, \text{timestamp}, \text{lifetime}, K_{c,s}\}K_s$

Since only Kerberos and the service share the private key K_s , the ticket is known to be authentic. The ticket contains a new private session key, $K_{c,s}$, known to the client as well; this key may be used to encrypt transactions during the session.³

Table 1: Notation

c	client principal
s	server principal
tgs	ticket-granting server
K_x	private key of "x"
$K_{c,s}$	session key for "c" and "s"
$\{info\}K_x$	"info" encrypted in key K_x
$\{T_{c,s}\}K_s$	Encrypted ticket for "c" to use "s"
$\{A_c\}K_{c,s}$	Encrypted authenticator for "c" to use "s"
addr	client's IP address

To guard against *replay attacks*, all tickets presented are accompanied by an *authenticator*:

$\{A_c\}K_{c,s} = \{c, \text{addr}, \text{timestamp}\}K_{c,s}$

This is a brief string encrypted in the session key and containing a timestamp; if the time does not match the current time within the (predetermined) clock skew limits, the request is assumed to be fraudulent.

For services where the client needs bidirectional authentication, the server can reply with

$\{\text{timestamp} + 1\}K_{c,s}$

This demonstrates that the server was able to read *timestamp* from the authenticator, and hence that it knew $K_{c,s}$; that in turn is only available in the ticket, which is encrypted in the server's private key.

Tickets are obtained from the TGS by sending a *request*

$s, \{T_{c,tgs}\}K_{tgs}, \{A_c\}K_{c,tgs}$

In other words, an ordinary ticket/authenticator pair is used; the ticket is known as the *ticket-granting ticket*. The TGS responds with a ticket for server s and a copy of $K_{c,s}$, all encrypted with a private key shared by the TGS and the principal:

$\{\{T_{c,s}\}K_s, K_{c,s}\}K_{c,tgs}$

The session key $K_{c,s}$ is a newly-chosen random key.

The key $K_{c,tgs}$ and the ticket-granting ticket itself, are obtained at session-start time. The client sends a message to Kerberos with a principal name;

²The Project Athena Technical Plan [Mill87, section 2] describes a simpler threat environment, where eavesdropping and host impersonation are of primary concern. While this may be appropriate for MIT, it is by no means generally true. Consider, for example, a situation where general-purpose hosts also function as routers, and packet modification or deletion become significant concerns.

³Technically speaking, $K_{c,s}$ is a *multi-session key*, since it is used for all contacts with that server during the life of the ticket.

Kerberos responds with

$$\{K_{c,igs}, \{T_{c,igs}\}K_{igs}\}K_c$$

The client key K_c is derived from a non-invertible transform of the user's typed password. Thus, all privileges depend ultimately on this one key.

Note that servers must possess private keys of their own, in order to decrypt tickets. These keys are stored in a secure location on the server's machine.

THE KERBEROS ENVIRONMENT

The Project Athena computing environment consists of a large number of more or less anonymous workstations, and a smaller number of large autonomous server machines. The servers provide volatile file storage, print spooling, mailboxes, and perhaps some computing power; the workstations are used for most interaction and computing. Generally, they possess local disks, but these disks are effectively read-only; they contain no long-term user data. Furthermore, they are not physically secure; someone so inclined could remove, read, or alter any portion of the disk without hindrance.

Within this environment the primary need is for user-to-server authentication. That is, when a user sits down at a workstation, that person needs access to private files residing on a server. The workstation itself has no such files, and hence has no need to contact the server or even to identify itself.

This is in marked contrast to a typical UNIX system's view of the world. Such systems do have an identity, and they do own files. Assorted network daemons transfer files in the background, clock daemons perform management functions, electronic mail and news arrives, etc. If such a machine relied on servers to store its files, it would have to assert, and possibly prove, an identity when talking to these servers. The Project Athena workstations are neither capable nor in need of such; they in effect function as very smart terminals with substantial local computing power, rather than as full computer systems.⁴

What does this mean for Kerberos? Simply this: Kerberos is designed to authenticate the end-user — the human being sitting at the keyboard — to some number of servers. It is not a peer-to-peer system; it is not intended to be used by one computer's daemons when contacting another computer. Attempting to use Kerberos in such a mode can cause trouble.⁵

We make this statement for several reasons. First and foremost, typical computer systems do not have a secure key storage area. In Kerberos, a

plaintext key must be used in the initial dialog to obtain a ticket-granting ticket. But storing plaintext keys in a machine is generally felt to be a bad idea; [Morr79] if a Kerberos key that a machine uses for itself is compromised, the intruder can likely impersonate any user on that computer, by impersonating requests vouched for by that machine (i.e., file mounts or cron jobs).⁶ Additionally, the session keys returned by the TGS cannot be stored securely; of necessity, they are stored in some area accessible to root. Thus, if the intruder can crack the protection mechanism on the local computer — or, perhaps more to the point, work around it for some limited purposes — all current session keys can be stolen. This is less serious than a breach of the primary Kerberos key, of course, since session keys are limited in lifetime and scope; nevertheless, one does not wish these keys exposed.

This points out a second flaw when multi-user computers employ Kerberos, either on their own behalf or for their users: the cached keys are accessible to attackers logged in at the same time. In a workstation environment, only the current user has access to system resources; there is little or no need even to enable remote login to that workstation. There are many reasons for this; a consequence, though, is that the intruder simply cannot approach the safe door to try to pick its lock.⁷ Only when the legitimate user leaves can the attacker attempt to find the keys. But the keys are no longer available; Kerberos attempts to wipe out old keys at logoff time, leaving the attacker to sift through the debris. With a multi-user computer, on the other hand, an attacker has concurrent access to the keys if there are flaws in the host's security.

There are two other minor flaws in Kerberos directly attributable to the environment. First, there is some question about where keys should be cached. Since all of the Project Athena machines have local disks, the original code used /tmp. But this is highly insecure on diskless workstations, where /tmp exists on a file server; accordingly, a modification was made to store keys in shared memory. However, there is no guarantee that shared memory is not paged; if this entails network traffic, an intruder can capture these keys.

Finally, the Kerberos protocol binds tickets to IP addresses. Such usage is problematic on multi-homed hosts (i.e., hosts with more than one IP address). Since workstations rarely have multiple addresses, this feature — intended to enhance security — was not a problem at MIT. Multi-user hosts often do have multiple addresses, however, and cannot live with this limitation. This problem has been

⁴We regard this as a feature, not a bug.

⁵More precisely, Kerberos is not a *host-to-host* protocol. In Version 5, it has been extended to support user-to-user authentication. [Davi90]

⁶Recall that we are assuming here that the machine — and hence its superuser — needs an identity of its own.

⁷On Project Athena machines, remote access to most workstations is in fact disabled.

fixed in Version 5.

PROTOCOL WEAKNESSES

Replay Attacks

The Kerberos protocol is not as resistant to penetration as it should be. A number of weaknesses are apparent; the most serious is its use of an authenticator to prevent replay attacks.

The authenticator relies on use of a timestamp to guard against reuse. This is problematic for several reasons. The claim is made that no replays are likely within the lifetime of the authenticator (typically five minutes). This is reinforced by the presence of the IP address in both the ticket and the authenticator. We are not persuaded by this logic. An intruder would not start by capturing a ticket and authenticator, and then develop the software to use them; rather, everything would be in place before the ticket-capture was attempted. Let us consider two examples.

Some years ago, Morris described an attack based on the slow increment rate of the initial sequence number counter in some TCP implementations. [Morr85] He demonstrated that it was possible, under certain circumstances, to spoof one half of a preauthenticated TCP connection without ever seeing any responses from the targeted host. In a Kerberos environment, his attack would still work if accompanied by a stolen live authenticator, but not if a challenge/response protocol was used. Alternatively, an intruder may simply watch for a "mail-checking" session, wherein a user logs in briefly, reads a few messages, and logs out. A number of valuable tickets would be exposed by such a session, notably the one used to mount the user's home directory. Note that the lifetime of the authenticators — 5 minutes — contributes considerably to this attack.

Further, the proposed Version 5 of Kerberos anticipates alternative communication protocols in which such replays may be trivial to implement. If Kerberos is to be considered as a general-purpose utility, it must make few security-critical assumptions about the underlying network, and those must be explicit.

It has been suggested that the proper defense is for the server to store all live authenticators; thus, an attempt to reuse one can be detected. [Ste88] In fact, the original design of Kerberos required such caching, though this was never implemented. (While that is a feature of the implementation rather than of the protocol itself, a security feature is not very useful if it is too hard to implement.)

For several reasons, we do not think that caching solves the problem. First, on UNIX systems it is difficult for TCP-based [Post81] servers to store authenticators. Servers generally operate by forking a separate process to handle each incoming request.

The child processes do not share any memory with the parent process, and thus have no convenient way to inform it — and hence any other child servers — of the value of the authenticator used. There are a number of obvious solutions — pipes, authenticator servers, shared memory segments and the like — but all are awkward, and some even raise authentication questions of their own. To date, we know of no multi-threaded server implementation which caches authenticators.

UDP-based [Post80] query servers can store the authenticators more easily, as a single process generally handles all incoming requests; however, they might have problems with legitimate retransmissions of the client's request if the answer was lost. (UDP does not provide guaranteed delivery; thus, all retransmissions happen from application level, and are visible to the application.) Legitimate requests could be rejected, and a security alarm raised inappropriately. One possible solution would be for the application to generate a new authenticator when retransmitting a request; were it not for the other weaknesses of the authenticator scheme, this would be acceptable.

Secure Time Services

As noted, authenticators rely on machines' clocks being roughly synchronized. If a host can be misled about the correct time, a stale authenticator can be replayed without any trouble at all. Since some time synchronization protocols are unauthenticated, [Post83, Mill88] and hosts are still using these protocols despite the existence of better ones, [Mill89] such attacks are not difficult.

The design philosophy of building an authentication service on top of a secure time service is itself questionable. That is, it may not make sense to build an authentication system assuming an already-authenticated underlying system. Furthermore, while spoofing an unauthenticated time service may be a difficult programming task, it is not cryptographically difficult.⁸ Using time-based protocols in a secure fashion means thinking through all these issues carefully and making the appropriate synchronization an explicit part of the protocol. As Kerberos is proposed for more varied environments, its dependence on a secure time service becomes more problematic and must be stressed.

As an alternative, we propose the use of a challenge/response authentication mechanism. As is done today, the client would present a ticket, though without an authenticator. The server would respond with a nonce identifier encrypted with the session

⁸In some environments, programming is not even necessary. Low-powered fake WWV transmitters are not hard to build, and, if properly located, could easily block out the legitimate signal.

key $K_{c,s}$; the client would respond with some function of that identifier, thereby proving that it possesses the session key.

Such an implementation is not without its costs, of course. An extra pair of messages must be exchanged each time a ticket is used, which rules out the possibility of authenticated datagrams. More seriously, all servers must then retain state to complete the authentication process. While not a problem for TCP-based servers, this may require substantial modification to UDP-based query servers. (The complexity of managing outstanding challenges may be comparable to that needed to cache live authenticators — the trade-off is not between a stateful and a stateless protocol, but in managing two kinds of state.)

There is a significant philosophical difference between the two techniques, however. In the current Kerberos implementation, with its assumptions about the network environment, retained state is only necessary to enhance security. The challenge/response scheme, on the other hand, guarantees security in a more general environment, but requires retained state to function at all.

Instead of substituting challenge/response throughout, a possible compromise is to extend the protocol with a challenge/response option. This option could be used, for example, to authenticate the user in the initial ticket-granting ticket exchange and to access a time service.⁹ Subsequent client-server interactions could use the current time-based protocol. But synchronizing the servers remains a problem; not synchronizing them will lead to denial of service, and if they access the time service as a client, they must somehow obtain and store a ticket and key to authenticate it. (See above on storing keys in servers.) Given these complexities and possible weaknesses, it would seem reasonable to allow any service to insist on the challenge/response option.

Summarizing, we emphasize that the security of Kerberos depends critically on synchronized clocks. In essence, the Kerberos protocols involve mutual trust among four parties: the client, server, authentication server and time server.

Password-Guessing Attacks

A second major class of attack on the Kerberos protocols involves an intruder recording login dialogs in order to mount a password-guessing assault. When a user requests $T_{c,ts}$ (the ticket-granting ticket), the answer is returned encrypted with K_c , a key derived by a publicly-known algorithm from the user's password. A guess at the user's password can be confirmed by calculating K_c and using it to

decrypt the recorded answer. An intruder who has recorded many such login dialogs has good odds of finding several new passwords; empirically, users do not pick good passwords unless forced to. [Morr79, Gram84, Stol88]

We propose the use of exponential key exchange [Diff76] to provide an additional layer of encryption. Without describing the algorithm in detail, it involves the two parties exchanging numbers that each can use to compute a secret key. An outsider, not knowing how the numbers were calculated, cannot easily derive the key.

Such a use of exponential key exchange would prevent a passive wiretapper from accumulating the network equivalent of `/etc/passwd`. While exponential key exchange is normally vulnerable to active wiretaps, such attacks are comparatively rare, especially if dedicated network routers are used.

Apart from licensing issues — exponential key exchange is protected by a U.S. patent — using it has its costs. LaMacchia and Odlyzko [LaMa] have demonstrated that exchanging small numbers is quite insecure, while using large ones is expensive in computation time. Additionally, we have added extra messages to the login dialog, and imposed the requirement for considerable extra state in the server. Given the trend towards hiding even encrypted passwords on UNIX systems, and given estimates that half of all logins at MIT are used within a two-week period, the investment may be justifiable. Perhaps the best solution is to support this feature as a domain-specific option.

Even exponential key exchange will not prevent all password-guessing attacks. Depending on how carefully the Kerberos logs are analyzed, an intruder need not even eavesdrop. Requests for tickets are not themselves encrypted; an attacker could simply request ticket-granting tickets for many different users. An enhancement to the server, to limit the rate of requests from a single source, may be useful.

Alternatively, some portion of the initial ticket request may be encrypted with K_c , providing a minimal authentication of the user to Kerberos, such that true eavesdropping would be required to mount this attack. (As we are preparing this manuscript, just such a suggestion is being hotly debated on the Kerberos mailing list. We originally overlooked an alternative avenue for mounting a password-guessing attack. Clients may be treated as services, and tickets to the client, encrypted by K_c , may be obtained by any user. This capability has been suggested as the basis for user-to-user authentication and enhanced mail services. [Salt90] But any such scheme would seem to require repeated re-entry of the user's password, an inconvenience we suspect will not be tolerated. We would prefer to provide the same functionality by having clients register separate instances as services, with truly random

⁹This was suggested to us by Clifford Neuman.

keys. Keys could be supplied to the client by the *keystore*, described below.)

An alternative approach is a protocol described by Lomas, Gong, Saltzer, and Needham. [Loma89] They present a dialog with a server that does not expose the user to password-guessing attacks. However, their protocol relies on public-key cryptography, an approach explicitly rejected for Kerberos.

Spoofing Login

In a workstation environment, it is quite simple for an intruder to replace the `login` command with a version that records users' passwords before employing them in the Kerberos dialog. Such an attack negates one of Kerberos's primary advantages, that passwords are never transmitted in cleartext over a network. While this problem is not restricted to Kerberos environments, the Kerberos protocol makes it difficult to employ the standard countermeasure: one-time passwords.

A typical one-time password scheme employs a secret key shared between a server and some device in the user's possession. The server picks a random number and transmits it to the user. Both the server and the user (with the aid of the device) encrypt this number using the secret key; the result is transmitted back to the server. If the two computed values match, the user is assumed to possess the appropriate key.

Kerberos makes no provision for such a challenge/response dialog at login time. The server's response to the login request is always encrypted with K_c , a key derived from the user's password. Unless a "smart card" is employed that understands the entire Kerberos protocol, this precludes any use of one-time passwords.

An alternative (first suggested to us by T.H. Foregger) requires that the server pick a random number R , and use K_c to encrypt R . This value $\{R\}K_c$, rather than K_c , would be used to encrypt the server's response. R would be transmitted in the clear to the user. If a hand-held authenticator was in use, the user would employ it to calculate $\{R\}K_c$; otherwise, the login program would do it automatically.

Several objections may be raised to this scheme. First, hand-held authenticators are often thought to be inconvenient. This is true; however, they offer a substantial increase in security in high-threat environments. If they are not used, the cost of our scheme is quite low, simply one extra encryption on each end.

A second, more cogent, objection is that if the client's workstation cannot be trusted with a user's password, it cannot be trusted with session keys provided by Kerberos. This is, to some extent, a valid criticism, though we believe that compromise of the

login password is much more serious than the capture of a few limited-lifetime session keys. This problem cannot be solved without the use of special-purpose hardware, a subject we shall return to below.

Finally, it has been pointed out that a user can always supply a known-clean boot device, or boot via the network. The former we regard as improbable in practice unless removable media are employed; the latter is insecure because the boot protocols are unauthenticated.

Inter-Session Chosen Plaintext Attacks

According to the description in the Version 5 draft, [Kohl89] servers using the KRB_PRIV format are susceptible to a *chosen plaintext attack*. (A chosen-plaintext attack is one where an attacker may choose all or part of the plaintext and, typically, use the resulting cipher text to attack the cipher. Here we use the cipher text to attack the protocol. Mail and file servers are examples of servers susceptible to such attacks.) Specifically, the encrypted portion of messages of this type have the form

$X = (\text{DATA}, \text{timestamp} + \text{direction}, \text{hostaddress}, \text{PAD})$

Since cipher-block chaining [FIPS81, Davi89] has the property that prefixes of encryptions are encryptions of prefixes, if DATA has the form

$(\text{AUTHENTICATOR}, \text{CHECKSUM}, \text{REMAINDER})$

then a prefix of the encryption of X with the session key is the encryption of

$(\text{AUTHENTICATOR}, \text{CHECKSUM}),$

and can be used to spoof an entire session with the server.

It may be argued that most servers are not susceptible to chosen plaintext attacks. Given that there are easy counters to this attack, it seems foolish to advocate a general format for private servers that does not also protect against it.

It should be noted that the simple attack above does not work against Kerberos Version 4, in which the encrypted portion of the KRB_PRIV message is of the form

$(\text{length}(\text{DATA}), \text{DATA}, \text{msectime}, \text{hostaddress}, \text{timestamp} + \text{direction}, \text{PAD})$

as the leading $\text{length}(\text{DATA})$ field disrupts the prefix-based attack. We leave it to the reader to discover a more complicated chosen ciphertext attack against this format, even allowing for the fact that Version 4 uses the nonstandard PCBC mode of encryption. (Hint: assume the initial vector is fixed and public.) However, it is interesting to note that the order of concatenation of message fields can have security-critical implications. We return to this question in the later section on message encoding.

Exposure of Session Keys

The term "session key" is a misnomer in the Kerberos protocol. This key is contained in the service ticket and is used in the multiple sessions between the client and server that use that ticket. Thus, it is more properly called a "multi-session key". Making this point explicit leads naturally to the suggestion that true session keys be negotiated as part of the Kerberos protocol. This limits the exposure to cryptanalysis [Kahn67, Beke82, Deav85] of the multi-session key contained in the ticket, and precludes attacks which substitute messages from one session in another. (The chosen-plaintext attack of the previous section is one such example.) The session key could be generated by the server or could be computed as a session-specific function of the multi-session key.

The Scope of Tickets

Kerberos tickets are limited in both time and space. That is, tickets are usable only within the realm of the ticket-granting server, and only for a limited period of time. The first is necessary to the design of Kerberos; the TGS would not have any keys in common with servers in other realms. The latter is a security measure; the longer a ticket is in use, the greater the risk of it being stolen or compromised.

A further restriction on tickets, in Version 4, is that they cannot be forwarded. A user may obtain tickets at login time, and use these to log in to some other host; however, it is not possible to obtain authenticated network services from that host unless a new ticket-granting ticket is obtained. And that in turn would require transmission of a password across the network, in violation of fundamental principles of Kerberos's design.¹⁰

Version 5 incorporates provisions for ticket-forwarding; however, this introduces the problem of cascading trust. That is, a host *A* may be willing to trust credentials from host *B*, and *B* may be willing to trust host *C*, but *A* may not be willing to accept tickets originally created on host *C*, which *A* believes to be insecure. Kerberos has a flag bit to indicate that a ticket was forwarded, but does not include the original source.

A second problem with forwarding is that the concept only makes sense if tickets include the network address of the principal. If the address is omitted — as is permitted in Version 5 — a ticket may be used from any host, without any further modifications to the protocol. All that is necessary to employ such a ticket is a secure mechanism for

copying the multi-session key to the new host. But that can be accomplished by an encrypted file transfer mechanism layered on top of existing facilities; it does not require flag bits in the Kerberos header.

Is it useful to include the network address in a ticket? We think not. Given our assumption that the network is under full control of the attacker, no extra security is gained by relying on the network address. In fact, the primary benefit of including it appears to be preventing immediate reuse of authenticators from a different host.

Even with the protection provided by network addresses, replay attacks that involve faked addresses are easy; again, see [Morr85]. Furthermore, an attacker can always wait until the connection is set up and authenticated, and then take it over, thus obviating any security provided by the presence of the address. Given these problems, and the cascading trust issue raised earlier, we suggest that ticket-forwarding be deleted.

A new inter-realm authentication mechanism is also introduced in Version 5. Briefly, if a user wishes to access a service in another realm, that user must first obtain a ticket-granting ticket for that realm. This is done by making the ticket-granting server in a realm the client of another realm's TGS. It in turn may be a client of yet another realm's TGS. A user's ticket request is signed by each TGS and passed along; realms will normally be configured in a hierarchical fashion, though "tandem links" are permitted.

Unfortunately, this scheme, while appearing to solve the problem, is deficient in several respects. First, and most serious, there is no discussion of how a TGS can determine which of its neighboring realms should be the next hop. Moving up the tree, towards the root, is an obvious answer for leaf nodes; however, each parent node would need complete knowledge of its entire subtree's realms in order to determine how to pass the request downwards. There are obvious analogies here to network-layer routing issues; note, though, that any "realm routing protocol" must include strong authentication provisions.

Another answer is to say that static tables should be used. This, too, has its security limitations: should realm administrators rely on electronic mail messages or telephone calls to set up their routing tables? If such calls are not authenticated, the security risks are obvious; if they are, the security of a Kerberos realm is subordinated to the security of a totally different authentication system.

There is also an evident link between inter-realm authentication and the cascading-trust problem. Kerberos Version 5 attempts to solve this by including path information in the ticket request. However, in the absence of a global name space, it

¹⁰Actually, a special-purpose ticket-forwarder was built for Version 4. However, the implementation was of necessity awkward, and required participating hosts to run an additional server.

is not clear that this is useful. If a realm is not a neighbor, its name may not carry any global significance, whether by malice or coincidence. Furthermore, to assess the validity of a request, a server needs global knowledge of the trustworthiness of all possible transit realms. In a large internet, such knowledge is probably not possible.

KERBEROS HARDWARE DESIGN CRITERIA

A Host Encryption Unit

One of the major reasons we question the suitability of Kerberos for multi-user hosts is the need for plaintext key storage. What if the host were equipped with an attached cryptographic unit? We consider the design parameters for such a box.

The primary goal is to perform cryptographic operations without exposing any keys to compromise. These operations must include validating tickets presented by remote users, creating requests for both ticket-granting tickets and application tickets, and encrypting and decrypting conversations. Consequently, there must be secure storage for an adequate number of keys, and the operating system must be able to select which key should be used for which function.

The next question, of course, is how keys are entered into the secure storage area. If tickets are decrypted by the encryption box but transferred to the host's memory for analysis, the embedded session key is exposed.¹¹ Therefore, we conclude that the encryption box itself must understand the Kerberos protocols; nothing less will guarantee the security of the stored keys.

Entry of user keys is more problematic, since they must travel through the host. Unless user terminals are connected directly to the encryption unit, there is little choice. Storing them off the host, though, is a significant help, as the period of exposure is then minimized. Host-owned keys — service keys, or the keys that `root` would use to do NFS mounts — should be loaded via a Kerberos-authenticated service resident in the encryption unit. We shall return to this point below.

We must now ensure that the protocol itself does not provide a mechanism to obtain keys. Looking at the message definitions, we see that only session keys are ever sent, and these are always sent encrypted. Furthermore, user machines never generate any such messages; they merely forward them. Thus, the box need not have the ability to transmit a key, thereby providing us with a very high level of

assurance that it will not do so.

If an encryption box is used for the Kerberos server itself, the problem is somewhat more complex. There are two places where keys are transmitted. First, when a ticket is granted, the ticket itself contains a session key, and a copy of that session key is sent back encrypted in the client's ticket-granting session key. Second, during the initial dialog with Kerberos, the ticket-granting session key must be sent out, encrypted in the client's password key. Note, though, that permanent keys are *never* sent; again, this assures us that the encryption box will not give away keys. Furthermore, since these session keys are intended to be random, we can buy ourselves a great deal of security by including a hardware random number generator on-board.

We are not too concerned about having to load client and server keys onto the board. This operation is done only by the Kerberos master server, for which strong physical security must be assumed in any event. It is possible that such an encryption unit can be made sufficiently tamper-resistant that even workstations can use them; certainly, there are commercial cryptographic devices that claim such strengths.

One major objection to this entire scheme is that ultimately, the encryption box is controlled by the host computer. Thus, if `root` is compromised, the host could instruct the box to create bogus tickets. Such concerns are certainly valid. However, as noted above, we consider such temporary breaches of security to be far less serious than the compromise of a key. Furthermore, using a separate unit allows us to create untamperable logs, etc.

It is also desirable to prevent misuse of keys. For example, we do not want the login key used to decrypt the arbitrary block of text that just happens to be the ticket-granting ticket. Accordingly, keys should be tagged with their purpose. A login key should be used only to decrypt the ticket-granting ticket; the key associated with it should be used only for obtaining service tickets, etc. Since the encryption box is performing all of the key management, this is not a difficult problem.

The Key Storage Unit

A variety of technologies may be used to implement encryption units, ranging from special boards to dedicated microcomputers connected to server hosts by physically-secure lines. If the latter is used, there is the temptation to use its disk storage to hold the service keys associated with the attached host, but we feel that that is inadvisable. Any media of that sort must be backed up, and the backups must be carefully guarded. Such a high degree of security may be impractical in some environments. Instead, we suggest that keys be kept in volatile memory, and downloaded from a secure *keystore* on

¹¹This is not a hypothetical concern. A program to do just that (for conventional passwords) was posted to *netnews* as long ago as 1984. It operated by reading `/dev/kmem`. The existence of this program was a principal factor motivating the current restrictive permission settings on `/dev/kmem`.

request, via an encryption-protected channel. Thus, only one master key need be stored within the box; this key could either be in non-volatile storage, or be supplied by an operator when necessary.

More generally, the keystore is a secure, reliable repository for a limited amount of information. A client of the keystore could package arbitrary data to be retained by the keystore, and retrieved at a later date. This data — the service keys and tags, in the case of an encryption unit, or even a conventional Kerberos host — would be uninterpreted by the keystore. Storage and retrieval requests would be authenticated by Kerberos tickets, of course. Only encrypted transfer (KRB_PRIV) should be employed, as insurance against disclosure of such sensitive material.

As noted, the same keystore protocol could be used to supply additional keys for new instances of the same client. For example, a user *pat* could have a separate instance *pat.email*, for receiving encrypted electronic mail. The key for that instance would be restricted to that user, of course.

Generally, transactions with the keystore are initiated by the client. However, there is some question about how to create the additional user keys, as user workstations are not particularly good sources of random keys. The best alternative is to provide a (secure) random number service on the network. When a new client instance is added, this service would be consulted to generate the key; both Kerberos and the keystore would be told about the key.

SECURITY VALIDATION

Is Kerberos correct? By that we are asking if there are bugs (or trapdoors!) in the design or implementation of Kerberos, bugs that could be used to penetrate a system that relies on Kerberos. Some would say that by making the code widely available, the implementors have enabled would-be penetrators to gain a detailed knowledge of the system, thereby simplifying their task considerably. We reject that notion.

In the late nineteenth century, Kerckhoffs formulated the basic principal under which the security of cryptographic systems should be evaluated: all details of the system design should be assumed to be known by the adversary. Only cryptographic keys specifically assumed to be secret should be unavailable to an attacker. [Kahn67, Kerc83] Given this basic premise, the security of a cryptographic system is evaluated based on concerted efforts at cryptanalysis.

Kerberos is designed primarily as an authentication system incorporating a traditional cryptosystem (the Data Encryption Standard) as a component. Never the less, the philosophy guiding Kerckhoffs' evaluation criterion applies to the evaluation of the security of Kerberos. The details of Kerberos's

design and implementation must be assumed known to a prospective attacker, who may also be in league with some subset of servers, clients, and (in the case of hierarchically-configured realms) some authentication servers. Kerberos is secure if and only if it can protect other clients and servers, beginning only with the premise that these client and server keys are secret, and that the encryption system is secure. Moreover, in the absence of a central, trusted "validation authority", each prospective user of Kerberos is responsible for judging its security. Of course, a public discussion of system security and publication of security evaluations will facilitate such judgments.

By describing the Kerberos design in publications and making the source code publically available, the Kerberos designers and implementors at Project Athena have made a commendable effort to encourage just such a public system validation. Obviously, this document is itself part of that process. However, the system design and its implementation have undergone significant modification, in part as a consequence of this public discussion. We stress that each modification to the design and implementation results in a new system whose security properties must be considered anew. (Examples of such modifications are the incorporation of hierarchically-organized servers and forwardable tickets in Version 5.)

Hence, on-going modification of Kerberos makes it a moving target for security validation attempts. A detailed security analysis would thus be premature. However, the proposed changes to Kerberos in the next few sections are intended, not so much to defeat specific attacks, as to facilitate the validation process. In particular, these suggestions are intended to make Kerberos more modular, in design and implementation. Doing so should make the security consequences of modifications more apparent, and facilitate an incremental approach to Kerberos security validation.

Message Encoding and Cut-and-Paste Attacks

The most simple analysis of the security of the Kerberos protocols should check that there is no possibility of ambiguity between messages sent in different contexts. That is, a ticket should never be interpretable as an authenticator, or vice versa. Such an analysis depends on redundancy in the pre-encryption binary encodings of each of the ticket and authenticator information. Currently, that analysis must be repeated with every modification to the protocol. This repetitive and often intricate analysis would be unnecessary if standard encodings (such as ASN.1) [ASN1, BER] were used. These encodings should include the overall message type (such as KRB_TGS_REP or KRB_PRIV). Together with reasonable assumptions about the encryption layer (see the next section), such an encoding scheme would

greatly simplify the protocol validation process, particularly as the protocol is modified or extended.

Some use of ASN.1 encodings has been adopted for other reasons in Version 5. We reinforce here that there are design principles other than standards compatibility that motivate such a change.

The Encryption Layer

Version 4 of Kerberos uses the nonstandard PCBC mode of encryption, *propagating cipher block chaining*, in which plaintext block $i+1$ is exclusive-or'ed with both the plaintext and ciphertext of block i before encryption. This mode was observed to have poor propagation properties that permit message-stream modification: specifically, if two blocks of ciphertext are interchanged, only the corresponding blocks are garbled on decryption. Version 5 replaces PCBC mode with the standard CBC mode, *cipher block chaining*, which exclusive-or's just the ciphertext of block i with the plaintext of block $i+1$ before encryption. A checksum — as of Draft 2, the exact form had not been determined — is used to detect message modification. In order to ensure that duplicate messages have different encryptions, random initial "confounders" are added to some message formats. In addition, Version 5 supports alternative encryption algorithms as options.

Both the confounder and checksum mechanisms are meant to augment the security of CBC encryption. They belong in a separate encryption layer, not at the level of the Kerberos protocols themselves. Further, the confounder mechanism should be replaced by using the standard initial vector mechanism of cipher-block chaining. [FIPS81, Davi89]

To prevent message-stream modification during authenticated or private sessions, Version 5 uses a timestamp field to prevent entire encrypted messages from being replayed. This is another concern more properly delegated to the encryption layer, where chaining across the packets of the entire session is the more standard mechanism. (Such chaining avoids both the dependence on a clock and the need to cache recent timestamps.)

Separating the Kerberos protocols from the details of encryption would facilitate both validation of the security of the Kerberos protocols, and implementations and validations involving alternative cryptosystems. Too much focus on mechanism, while endemic to cryptographic protocol design, leads away from the need to state the basic properties required of the encryption layer. We would suggest the following adversarial analysis as the starting point for such a specification: allow an adversary to submit, one after the other, any number of messages for encryption under an unknown key K . The adversary also has the ability to take prefixes and suffixes of known messages, exclusive-or known messages, and encrypt or decrypt with known keys. At the end

of this process, the adversary should not be able to produce any encrypted messages other than those specifically submitted for encryption. Such an analysis would preclude encryption schemes susceptible to simple chosen-plaintext attacks, as described in a previous section.

Given the intractability of reasoning about DES, or of proving complexity properties of any cryptosystem with bounded key size, such analyses will be no guarantee of overall security. But they can be used to preclude the existence of trivial cut-and-paste attacks. [DeMi83, Moor88]

RECOMMENDED CHANGES TO THE KERBEROS PROTOCOL

Below, we list our recommended changes to the Kerberos protocol. Our ranking is governed by our estimate of the likelihood and consequences of the attack, balanced against the difficulty of implementing the modification.

1. A challenge/response protocol should be offered as an optional alternative to time-based authentication.
2. Use a standard message encoding, such as ASN.1, which includes identification of the message type within the encrypted data.
3. Alter the basic login protocol to allow for handheld authenticators, in which $\{R\}K_C$, for a random R , is used to encrypt the server's reply to the user, in place of the key K_C obtained from the user password. This allows the login procedure to prompt the user with R , who obtains $\{R\}K_C$ from the handheld device and returns that value instead of the password itself.
4. Mechanisms such as random initial vectors (in place of confounders), block chaining and message authentication codes should be left to a separate encryption layer, whose information-hiding requirements are clearly explicated. Specific mechanisms based on DES should be validated and implemented.
5. The client/server protocol should be modified so that the multi-session key is used to negotiate a true session key, which is then used to protect the remainder of the session.
6. Support for special-purpose hardware should be added, such as the keystore. More importantly, future enhancements to the Kerberos protocol should be designed under the assumption that a host, particularly a multi-user host, may be using encryption and key-storage hardware.
7. To protect against trivial password-guessing attacks, the protocol should not distribute tickets for users (encrypted with the password-based key), and the initial exchange should authenticate the user to the Kerberos server.
8. Support for optional extensions should be

included. In particular, an option to protect against password-guessing attacks via eavesdropping may be a desirable feature.

ACKNOWLEDGEMENTS

We would like to thank D. Davis and T.H. Foregger for their comments on an early draft. We'd especially like to thank C. Neuman for his detailed reviews of many versions of the paper, and his willingness to discuss the issues with us. W. Griffith helped us with preparation of the appendix on Draft 3. Finally, we'd like to thank the Project Athena and Kerberos development staff for their initial design and implementation of Kerberos, their solicitation of comments, and their responsiveness to our criticisms.

References

- FIPS81. , "DES Modes of Operation," Federal Information Processing Standards Publication 81 (December 1980). National Bureau of Standards, U.S. Department of Commerce
- ASN1. , "Information Processing Systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)," International Standard 8824 (1987). International Organization for Standardization and International Electrotechnical Committee
- BER. , "Information Processing Systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)," International Standard 8825 (1987). International Organization for Standardization and International Electrotechnical Committee
- Beke82. H. Beker and F. Piper, *Cipher Systems*, John Wiley & Sons (1982).
- Brya88. B. Bryant, *Designing an Authentication System: A Dialogue in Four Scenes*, Draft February 8, 1988.
- Davi89. D.W. Davies and W.L. Price, *Security for Computer Networks*, John Wiley & Sons (1989). Second Edition
- Davi90. D. Davis and R. Swick, *Workstation Services and Kerberos Authentication at Project Athena*, MIT Laboratory for Computer Science Technical Memorandum 424 (February 1990).
- Deav85. C.A. Deavours and L. Kruh, *Machine Cryptography and Modern Cryptanalysis*, Artech House (1985).
- DeMi83. R. DeMillo and M. Merritt, "Protocols for Data Security," *Computer* 16(2) pp. 39-50 (February 1983).
- Diff76. W. Diffie and M.E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory* 6 pp. 644-654 (November, 1976).
- Gram84. F.T. Grampp and R.H. Morris, "Operating System Security," *AT&T Bell Laboratories Technical Journal* 63(8, Part 2) pp. 1649-1672 A&T, (October, 1984).
- Kahn67. D. Kahn, *Codebreakers: The Story of Secret Writing*, Macmillan (1967).
- Kerc83. A. Kerckhoffs, *La Cryptographie Militaire*, Librairie Militaire de L. Baudoin & Cie., Paris (1883).
- Kohl89. J. Kohl, B. Clifford Neuman, and J. Steiner, *The Kerberos Network Authentication Service*, MIT Project Athena (November 6, 1989). Version 5, Draft 2
- Kohl90. J. Kohl, B. Clifford Neuman, and J. Steiner, *The Kerberos Network Authentication Service*, MIT Project Athena (October 8, 1990). Version 5, Draft 3
- LaMa. B.A. LaMacchia and A.M. Odlyzko, *Computation of Discrete Logarithms in Prime Fields*, (Manuscript in preparation)
- Loma89. T.M.A. Lomas, L. Gong, J.H. Saltzer, and R.M. Needham, "Reducing Risks from Poorly Chosen Keys," *Operating Systems Review* 23(5) pp. 14-18 ACM, (December 1989).
- Mill87. S.P. Miller, B.C. Neuman, J.I. Schiller, and J.H. Saltzer, "Kerberos Authentication and Authorization System," in *Project Athena Technical Plan*, (December 1987). Section E.2.1
- Mill88. D.L. Mills, "Network Time Protocol," RFC 1059 (July 1988).
- Mill89. D.L. Mills, "Network Time Protocol," RFC 1119 (September 1989).
- Moor88. J.H. Moore, "Protocol Failures in Cryptosystems," *Proc. IEEE* 76(5) pp. 594-602 (May 1988).
- Morr79. R. Morris and K. Thompson, "UNIX Password Security," *Communications of the ACM* 22(11) p. 594 (November 1979).
- Morr85. R.T. Morris, "A Weakness in the 4.2BSD TCP/IP Software," Computing Science Technical Report No. 117, AT&T Bell Laboratories, Murray Hill, New Jersey (February 1985).
- Post80. J.B. Postel, "User Datagram Protocol," RFC 768 (August 28, 1980).
- Post81. J.B. Postel, "Transmission Control Protocol," RFC 793 (September 1981).
- Post83. J.B. Postel and K. Harrenstien, "Time Protocol," RFC 868 (May 1983).
- Rive90. R.L. Rivest, "MD4 message digest algorithm," RFC 1186 (October 1990).
- Salt90. J.H. Saltzer, private communication June 19, 1990.

- Stein88. J. Steiner, C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," in *Proc. Winter USENIX Conference*, , Dallas (1988).
- Stoll88. C. Stoll, "Stalking the Wiley Hacker," *Communications of the ACM* 31(5) p. 484 (May 1988).
- Voyd83. V.L. Voydock and S.T. Kent, "Security Mechanisms in High-Level Network Protocols," *ACM Computer Surveys* 15(2) pp. 135-171 (June, 1983).

Steven M. Bellovin received a B.A. degree from Columbia University, and an M.S. and Ph.D. in Computer Science from the University of North Carolina at Chapel Hill. While a graduate student, he wrote the original version of *pathalias* and helped create *netnews*. However, the former is not an indictable offense, and the statute of limitations on the latter has expired. Nevertheless, he is still atoning for both actions. He has been at AT&T Bell Laboratories since 1982, where he does research in networks, security, and why the two don't get along. He may be reached electronically as smb@ulysses.att.com; those who prefer to murder trees may send scraps of paper to Room 3C-536B, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.



Michael Merritt received a B.S. from Yale University, and an M.S. and Ph.D. in Information and Computer Science from the Georgia Institute of Technology. His dissertation, "Cryptographic Protocols", developed techniques for exploring security properties of distributed algorithms. He has been at AT&T Bell Laboratories since 1983, where he does research in distributed systems and security. His email address is mischu@research.att.com; paper to Room 3D-458, AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, U.S.A.



APPENDIX: VERSION 5 DRAFT 3

Draft 3 has gone a long way towards alleviating our concerns. Many problems have been fixed, and provisions have been made for compatible enhancements to resolve other outstanding issues. These are being refined in ongoing discussion. Still, some issues remain unresolved or unaddressed. In addition, we raise new issues related to older areas

of the specification.

In a few places, we mention changes that may be made in future revisions of the specification; the reader is cautioned that these represent our understanding, and only our understanding, of a continuing process.

With one exception, this summary omits areas where the authors' intent was clear or was clarified in private communications. That exception — a way to misuse weak checksums to subvert bidirectional authentication — we include to demonstrate the delicacy inherent in the design and specification of authentication protocols.

Draft 3 and Our Recommended Changes

We begin by reviewing our recommended changes in light of Draft 3 and subsequent discussions with its authors.

1. The `KRB_AS_REQ/KRB_AS_REP` and `KRB_TGS_REQ/KRB_TGS_REP` exchanges now provide challenge/response authentication of the server to the client via a *nonce* field, instead of depending on the workstation time. For application servers, the *e-data* field in the `KRB_AP_ERR_METHOD` error message can be used by the server to signal the client to use a challenge/response alternative to the time-based kerberos authentication.
2. All encrypted data is labeled with the message type prior to encryption, via full integration of the ASN.1 standard. Although there were many reasons for this decision, we applaud its beneficial impact on security.
3. An optional *padata* field will probably be added to the `KRB_AS_REP` to allow for handheld authenticator protocol extensions.
4. As discussed, mechanisms such as random initial vectors (in place of confounders), block chaining and message authentication codes are now left to a separate encryption layer, with a much clearer discussion of requirements and of specific mechanisms based on DES.
5. Optional fields will probably be added to the `AP_REQ` and `AP_REP` messages to support the negotiation of true session keys.
6. Addition of optional fields (such as *padata*) should facilitate extensions that exploit special-purpose hardware.
7. The initial exchange still does not authenticate the user to the Kerberos server. Thus, the Kerberos equivalent of `/etc/passwd` must be treated as public, and passwords must be chosen and administered with password-guessing attacks in mind. However, the *padata* field facilitates optional implementation of such preauthentication mechanisms.
8. As above, several optional fields facilitate extensions such as exponential-key exchange to protect against password-guessing via

eavesdropping.

The following sections discuss some of the revisions in Draft 3 in more detail, and raise some new issues.

Login Dialog

The login dialog has been enhanced to include an additional authentication data field. This can be used to support hand-held authenticators, pre-encryption of the original request, and future extensions. This is a significant enhancement, but we regret that support for hand-held authenticators and pre-encryption is not yet a part of the standard.

In particular, the optional field in the request message can support some sort of pre-encryption. For example, the nonce field can be sent both in the clear and encrypted in the user's login key, thereby demonstrating that the client is legitimate, and precluding remote collection of tickets encrypted with the user's key. As discussed in the main body of this paper, we feel such a mechanism should be mandatory, not optional. Password-cracking programs require just this sort of data; there is no need to provide grist for their mill.

As currently released, a challenge-response dialog cannot be implemented by the Draft 3 reply format. While the request message possesses the optional extra field, the reply does not, and hence cannot carry the encrypted key. Adding this field would also permit compatible support of exponential key exchange, wherein each party must send a random exponential. We understand that the optional field will probably be added to the reply.

The Encryption and Checksum Layers

There is now a separate, well-defined encryption layer, with specified properties. Among these are that the encryption module be capable of detecting any tampering with the message. The only supported method, in this version, is a CRC-32 checksum sealed within the encrypted portion of the message.

The encryption layer also reaps the benefit of the ASN.1 encoding. Since the encoding includes a length field, it is no longer possible for an attacker to truncate a message, and present the shortened form as a valid encrypted message. If a decision were ever made to replace ASN.1 (say, with something more efficient), this property would need to be preserved.

The confounder has now been moved to the encryption layer, but there is still some confusion of function with the IV used by CBC-mode encryption. As commonly used, an IV is a confounder (see, for example, [Voyd83]); to hold it constant during a session negates its purpose and thus requires the additional confounder. We suggest that the IV be used as intended, and be incremented or otherwise altered after each message. Initial values for it should be

exchanged during (or derived from) the authentication handshake. Apart from simplifying the definition of the encryption function, this scheme would also allow detection of message deletions by interested applications.

It could be argued that requiring the IV to be handled at a higher layer violates the layering we have espoused. However, an IV is as much an attribute of a cryptosystem as is a key. It would be reasonable to encapsulate the definition of the IV into the definition of the key object passed down to the encryption layer.

The properties required of checksums are not as well-defined. Three types are specified: CRC-32, MD4 and MD4 encrypted with DES. [Rive90] However, no mention is made of their attributes, save that some are labeled "cryptographic". This is a crucial omission, as discussed below. A better classification is whether or not a checksum is "collision-proof", that is, whether or not an attacker can construct a new message with the same checksum. The CRC-32 checksum is not collision-proof, while MD4 is believed to be. Note that encrypting a checksum provides very little protection; if the checksum is not collision-proof and the data is public, an adversary can compute the value and replace the data with another message with the same checksum value. (Several such attacks are indicated below.)

Weak Checksums and Cut-and-Paste Attacks

One of the major changes in Draft 3 was the removal of encryption protection from the additional tickets and authorization data that may be enclosed with certain requests. These fields are protected by a checksum sealed in the encrypted authenticator sent with the request. Assume that the checksum algorithm used is CRC-32. (This is permitted by a literal reading of Draft 3, though we have learned that this was not the intent of the authors.) With this assumption, the existence of the ENC-TKT-IN-SKEY option leads to a major security breach, and in particular to the complete negation of bidirectional authentication.

As usual, the client, possessing a valid ticket-granting ticket, sends off a request for a new ticket for some service *S*. The enemy intercepts this request and modifies it. First, the ENC-TKT-IN-SKEY bit is set. This specifies that the ticket, normally encrypted in *S*'s key, should be encrypted in the session key of the enclosed ticket-granting ticket. Second, the attacker's own ticket-granting ticket is enclosed. Obviously, the attacker knows its session key. Finally, the additional authorization data field is filled in with whatever information is needed to make the CRC match the original version.

Consider what happens. The ticket-granting service, seeing a valid request, sends back a ticket. This ticket, encrypted in the enemy's key, will not be intelligible to the real service, but of course, it will not get that far. The legitimate client cannot tell that the ticket is misencrypted; tickets are, almost by definition, encrypted in a key known only to the server and Kerberos. When the service is requested, the enemy intercepts the request and unseals the ticket. The client may request bidirectional authentication; however, since the attacker has decrypted the ticket, the session key for that service request is available. Consequently, the bidirectional authentication dialog may be spoofed without trouble.

A number of different factors interacted to make this attack possible. One is obvious: the ticket request was protected by what turned out to be a weak checksum. If a collision-proof checksum were used, the attack would be infeasible; the enemy could not have generated the additional authorization data field necessary to make the new request's checksum match the original. But there are subtleties here. First, if the additional tickets used by ENC-TKT-IN-SKEY were encrypted (again), they would have been adequately protected by the very same CRC-32 checksum that was abused in the attack. However, because of the encryption, the enemy would be unable to either discern or match the checksum. In other words, the context is critical; merely refraining from re-encrypting some encrypted data, while using the same checksum to protect it, has led to a security breach. (Note: we have been told that the designers intended to require that the *cname* in the additional ticket match the name of the server for which the new ticket is being requested. This requirement would still permit the intended use of the option, but would foil the attack we describe. Apparently, the requirement was inadvertently omitted from Draft 3.)

A similar attack may be possible using the REUSE-SKEY option. This option was designed for multicast key distribution; with a weak checksum, an attacker can abuse it to generate a service ticket whose key is known. The REUSE-SKEY option also permits a related, albeit less serious, attack. If two tickets, *T1* and *T2*, share the same key, the attacker can intercept a request for one service, and redirect it to the other. Since the two tickets share the same key, the authenticator will be accepted. Just how damaging this possibility is depends on what sorts of services might want to share the same key. If, say, a file server and a backup server were invoked this way, an attacker might redirect some requests to destroy archival copies of files being edited. A solution to this particular attack is to include either the service name, a collision-proof checksum of the ticket, or both, in the authenticator. To be sure, Draft 3 explicitly warns against using

tickets with DUPLICATE-SKEY set for authentication. Servers that obey this restriction are not vulnerable to this attack. Also, we have been told that the REUSE-SKEY option will probably be omitted in future revisions of the protocol.

A last attack of this sort can occur if the attacker substitutes a different ticket for the legitimate one in key distribution replies from Kerberos. The encrypted part of such a message does not contain any checksum to validate that the message was not tampered with in transit. While this appears to be more a denial-of-service attack than a penetration, it would be useful for the client to know this immediately.

Two issues underly this list of potential attacks. As discussed, weak checksums (encrypted but not collision-proof, and over public data) allow an adversary to paste together legitimate-looking messages. Message integrity via strong checksums and/or encryption should be extended to as many protocol messages (and as many fields) as possible.

Second, the REUSE-SKEY and ENC-TKT-IN-SKEY options "overload" the basic protocol, in that tickets may now share session keys or be encrypted in keys other than the service. It is possible that there are other ways an attack could exploit the ensuing ambiguities. These options are intended for very constrained uses, not general authentication; they should not be so intimately integrated into the basic authentication protocol. The same purposes would be served by adding separate message types that cannot be misinterpreted as tickets, and using keys that are derived from but are not identical to those used in the basic protocol.

Even then, an analysis of the final standard is needed, to assure that a minor extension has not negated a security-critical assumption. (E.g., the basic Kerberos protocol assumes that no two tickets share a session key, and that tickets are always encrypted with the server's key.)

KRB_SAFE and KRB_PRIV Messages

The KRB_SAFE and KRB_PRIV messages employ the session key distributed with the ticket for integrity-checking and privacy, respectively. Draft 3 dictates that both use time-of-day values to guard against replay, which may be problematic. Currently, the resolution of the timestamp is limited to 1 millisecond, which is far too coarse for many applications. (This and other timestamps in the protocol will probably be changed to microsecond resolution.)

A second problem area is the need for a cache of recently-used timestamps. Obviously, if such messages are used for things like file system requests, the size of the cache could rapidly become unmanageable. Furthermore, if two authenticated or encrypted sessions run concurrently, the cache must

be shared between them, or messages from one session can be replayed into the other.

Both problems can be solved if the idea of a timestamp is abandoned in favor of sequence numbers. A random initial sequence number can be transmitted with the authenticator and/or in the KRB_AP_REP message; after each authenticated message is sent, it would, of course, be incremented. The cache is then a simple last-message counter. This mechanism also provides the ability to detect deleted messages, by watching for gaps in sequence number utilization. And, since each session would have its own initial sequence number, it would not be possible for an attacker to perform cross-stream replays, and concurrent access to a common cache is not necessary. (This advantage would be gained even with timestamps if true session keys were used.) It is likely that in a future revision, sequence numbers will be provided as an alternative to the use of timestamps.

Authenticators

Draft 3 still calls for the use of authenticators to guard against ticket replay. However, there is now a provision for the server to specify that additional authentication is required, and an optional data field for this has been added to the KRB_ERROR reply message. This can be used to implement challenge/response schemes.

The authenticator should have some other fields added to it, some of them optional. As noted earlier, it must contain a collision-proof checksum linking it to the ticket, and an optional initial sequence number. The latter would be used by any applications that might wish to exchange encrypted or authenticated messages.

The authenticator is also the right place to negotiate a true session key. We propose adding a new field for it to both the authenticator and the KRB_AP_REP message. The actual session key could be formed by an exclusive-or of the multisession key associated with the ticket, a randomly-generated field in the authenticator, and a similar field in the reply message. Note that this retains a measure of compatibility with the current scheme: if the two optional fields are not present, the multi-session key will be used as the actual session key.

Negotiation of true session keys, initial sequence numbers, and confounders or IV's could be combined in one standard mechanism, perhaps subsumed as encryption-specific subfields of the session key fields.

Inter-Realm Authentication

Inter-realm authentication is still problematic. Granted that static configuration files can tell a Kerberos server who its parent is, and even the identities of all of its children, there is still no scalable

mechanism to learn of grandchildren or more distant descendants.

To be sure, it is apparently the intention of the authors that the Internet's domain name space be used to denote realms, and — implicitly — the hierarchy of servers. It is far from clear to us that the two hierarchies coincide. Furthermore, such usage is not required. No alternative routing mechanism has been suggested.

Additionally, there are several pieces of the protocol that are unclear or simply do not work with inter-realm tickets. For example, ENC-TKT-IN-SKEY and REUSE-KEY require the ticket-granting server to decrypt a ticket. It cannot do this if the ticket had been issued by another realm. Presumably, of course, the request could be sent to the other realm's ticket-granting server, but it may not possess the necessary key to generate the new ticket.

NEW RECOMMENDED CHANGES

Below, we include a new list of recommended changes, beyond those we have indicated are likely to be adopted. The first two are repeated from our earlier list, and are now (or will be) implementable as options; we repeat them here to stress our belief that they should be a mandatory part of the protocol.

1. Alter the basic login protocol to allow for challenge/response handheld authenticators.
2. The initial exchange should authenticate the user to the Kerberos server, to complicate password-guessing attacks.
3. Strong checksums, encryption, and additional fields should be used to assure integrity of the basic Kerberos messages. (For example, tickets should be tied more closely to the contexts in which they are used, by including service names in the ticket, and the encrypted part of KRB_AS_REP and KRB_TGS_REP should contain collision-proof checksums of the tickets.)
4. Protocol extensions not related to basic authentication (the ENC-TKT-IN-SKEY and REUSE-SKEY options) should be omitted or use distinct message and ticket formats.

UNIX Password Encryption Considered Insecure

Philip Leong – University of Sydney
Chris Tham – State Bank of Victoria

ABSTRACT

Recently there has been a revival of interest in the security of the password encryption scheme employed in the UNIX Operating System and its derivatives. This resurgence was due mainly to the success of an attack on the Internet by a virus program in November 1988. The current encryption scheme used is a variant of the NBS Data Encryption Standard (DES) modified in such a way that existing DES hardware implementations cannot be used. There is currently no reported way of reversing the password encryption, i.e., to obtain a password from its encrypted string.

In this paper, we show that the current encryption scheme can no longer be considered secure as most UNIX passwords can be decrypted using a brute force search within a reasonable period of time. As an example, all passwords containing only lower case alphabetic characters can be decrypted in less than 15 days.

In order to perform a brute force search, we need the ability to encrypt a UNIX password in the shortest time possible. Accordingly, we present a hardware design of a password encryption device that can encrypt a UNIX password in 6 μ s. This device consists of approximately 100 Emitter Coupled Logic (ECL) chips and can be built by any electronic hobbyist for less than \$2000. The board can also be used to encrypt DES at 266 Mbps, more than ten times faster than a recent CMOS VLSI design.

We also present a software only implementation of the encryption algorithm recoded for maximum speed. This implementation can encrypt a UNIX password in 1.2 ms on an IBM RS/6000 Model 530 machine.

INTRODUCTION

The issue of the security of the UNIX Operating System has long been a subject of debate, resulting in a multitude of conflicting statements made, often by ill-informed parties. Whilst it is probably true that the system is "... more secure than any other operating system offering comparable facilities"[Duf89a] it is also true that UNIX has never been designed with security features foremost in its implementors' minds.[Rit78a]

The UNIX protection model has been extensively described in existing literature [Rit78a, Gra84a, Bac86a] and will not be detailed in this paper. It is sufficient only to know that UNIX security is based on the concept of *users* and *groups*. A user is a uniquely identifiable entity within the system and belongs in one or more groups. Users may own resources in the system such as processes, files and devices. Access to these resources is maintained by the owner who can control access by other users or a specific group. To gain access to a UNIX system, all users must undergo a login procedure either explicitly or implicitly. The login procedure requires the knowledge of a valid *login name* and a *password* associated with the user. The password is used to authenticate the user. (An implicit login occurs

when accessing a machine through a network using utilities which may bypass the interactive login procedure.)

Once through the login procedure, access to resources in the system is validated through the user/group protection mechanism. In addition, there exists a special user known as the *super-user* who has the ability to transcend the protection scheme to access any resource in the system.

Although there are various ways of compromising the security of a UNIX system, nearly all of them involve either (a) an unauthorized person or program gaining access to the system by knowing the password associated with an authorized user, or (b) an authorized user 'masquerading' as another user, preferably the super-user, in order to gain access to unauthorized resources in the system.¹ In any case, any intrusions into a system or network must first

¹It must be pointed out that unauthorized access to a system can happen through a network by compromising the network utilities. When this occurs, we classify it as an unauthorized access into a system by an authenticated user of *another* system. Also, the above classification scheme has not considered the possibility of an unauthorized user gaining access to the system simply by walking into a terminal already logged in.

begin with the knowledge of the password of an existing authorized user. Hence, the security of a UNIX system hinges on the security of its password authentication scheme.

On November 2, 1988, a self replicating program was released on the Internet (a logical network of many physical networks of predominantly UNIX machines) which uses the resources of machines on the network to replicate and spread itself. This program, alternately described as the *Internet Worm* and *Internet Virus*, [Eic89a, See89a, Spa88a] caused a major disruption to the operation of the Internet and incensed the members of the Internet computing community comprising thousands of academic, corporate and government users. It sparked our interest in investigating the effectiveness of the UNIX password encryption system as one of major methods of attack employed by the program involved the guessing of UNIX passwords through repeated executions of a password 'cracking' routine in the program. The implementation of the encryption routine used by the program was different from that used by the UNIX system itself and is up to nine times faster than the UNIX version. [See89a]

DESCRIPTION OF THE UNIX PASSWORD ENCRYPTION ALGORITHM

The *login(1)* program in the UNIX System implements the login procedure and attempts to authenticate access to the system. A file called */etc/passwd* contains a list of all valid users on the system, including their login names and encrypted passwords. This file, strangely enough, is readable by any user on the system. When a user tries to gain access to the system, she or he must first type in a valid login name. The *login* program prompts for a password associated with the login name. This password is not echoed back to the user as it is typed in. Once typed, the program calls a standard UNIX library function called *crypt(3)* which encrypts the password into a printable ASCII string. The *login* program then compares the results of the encryption with the encrypted password in */etc/passwd*. If the two strings are equal, the user is allowed entry into the system and the program then sets up the user environment and executes the user's command interpreter on behalf of the user. If the login name that has been typed in does not match a valid user on the system, or if the encrypted password does not match the encrypted string, the program prints the string "Login incorrect" and redisplay the login prompt.

Obviously, the design and implementation of the *crypt()* function is crucial to the security of the login procedure. The encryption performed by *crypt()* must be irreversible, i.e., it should be impossible to derive the clear password string given the encrypted form of the string, even when the source

to the encryption routine is available.² In addition, the encryption algorithm must be reasonably compact, given the hardware limitations of the machine on which UNIX was originally designed for, and yet take up a substantial amount of computing time to execute. This last requirement serves to prevent the use of key search cryptanalytic approaches.

The original implementation of the encryption algorithm was a variant of the M-209 cipher machine.³ The password was used as the key for the encryption of a constant text string and the result of the encryption was returned. Morris & Thompson [Mor78a] notes that a version of this algorithm optimized for maximum speed could encrypt a password in approximately 1.25 ms on a DEC PDP-11/70 minicomputer. This was considered unacceptably fast as it permitted the use of key search techniques in password guessing programs.

$$\begin{aligned} K &= k_1 k_2 \cdots k_{64} \\ PC1(K) &= C_0 D_0 \\ &= c_1 c_2 \cdots c_{28} d_1 d_2 \cdots d_{28} \\ C_i &= LS_i(C_{i-1}) \\ D_i &= LS_i(D_{i-1}) \\ K_i &= PC2(C_i D_i) \\ \text{where } i &= 1, 2, \dots, 16 \end{aligned}$$

Table 1: Computing the Key Schedule

The version currently in use is based on the Data Encryption Standard (DES) announced by the National Bureau of Standards (NBS) for use in unclassified United States Government applications in 1977. [Ano77a, FIP75a, Seb89a] The DES uses an algorithm called the Data Encryption Algorithm (DEA) specified in the American National Standard ANSI X3.92-1981. [ANS81a] The first eight characters of the user's password are used as the DES key, a constant 64-bit block (consisting of all zero bits) is then encrypted via DEA 25 times (the result of each encryption being used to feed the next round). Finally, the resultant 64-bits is converted into a string of 11 printable ASCII characters by encoding every six bits into a printable ASCII character and zero padding the 11th character.

The DEA is a fairly convoluted series of bit permutations, expansions and selections optimized for efficient hardware, rather than software, implementation. It requires a 64-bit key to be used to encrypt every 64-bit block to a 64-bit encrypted block.

The key K is only effectively 56-bits long as every eighth bit is ignored by the algorithm. K is used to compute a key schedule of 16 48-bit subkeys

²This additional requirement was necessary because the source code to the UNIX operating system was widely available within the academic community and also described in easily available literature.

³U.S. Patent Number 2,089,603

(K_1 to K_{16}). A permuted choice (PC1) function transforms K into two equal 28-bit halves (C_0 and D_0). These halves are rotated independently by specified amounts (LS_i) and then run through another permuted choice (PC2) yielding the 16 48-bit keys. Table 1 summarizes the key schedule computation.

The actual encryption algorithm itself will encrypt a 64-bit block T into another 64-bit block Z . T undergoes an initial permutation called IP which is then splitted into two equal halves called L_0 and R_0 . Each half is then alternately passed through the f function which expands the half into a 48-bit block through the E expansion, bitwise exclusive-ORs (\oplus) with one of the subkeys in the key schedule (K_i), performs selection (S) and permutation (P) operations before exclusive-ORing the 32-bit result with the other half. After 16 applications of the f function, the halves are then rejoined back into a 64-bit block and the result undergoes a final permutation (FP) yielding the encrypted block Z . Table 2 summarizes the operation of DEA.

$$\begin{aligned}
 T &= t_1 t_2 \cdots t_{64} \\
 T_0 &= IP(T) \\
 &= L_0 R_0 \\
 &= l_1 l_2 \cdots l_{32} r_1 r_2 \cdots r_{32} \\
 L_i &= R_{i-1} \\
 R_i &= L_{i-1} \oplus f(R_{i-1}, K_i) \\
 i &= 1, 2, \dots, 16 \\
 Z &= FP(R_{16} L_{16}) \\
 f(R, K) &= P(S(E(R) \oplus K))
 \end{aligned}$$

Table 2: DEA operation

One interesting twist in the implementation of the DES algorithm in the UNIX `crypt()` function lies in the *salting* of the encryption. Stored together with the encrypted password is a 12-bit salt encoded as two printable ASCII characters. The `crypt()` function expects the salt to be passed to it along with the clear password text. The salt (Ψ) is used to perturb the E expansion in the following manner. Let E be the standard expansion function and E' be the perturbed expansion function. Then $Y=E(X)$ and $Y'=E'(X)$ is related:

$$\begin{aligned}
 Y' &= y'_1 y'_2 \cdots y'_{48} \\
 Y &= y_1 y_2 \cdots y_{48} \\
 \Psi &= s_1 s_2 \cdots s_{12} \\
 y'_i &= \begin{cases} y_i & \text{if } s_i = 0 \\ y_{i+24} & \text{if } s_i = 1 \end{cases} \\
 y'_{i+24} &= \begin{cases} y_{i+24} & \text{if } s_i = 0 \\ y_i & \text{if } s_i = 1 \end{cases} \\
 i &= 1, 2, \dots, 12
 \end{aligned}$$

Table 3: Effect of the salt on the E expansion

When a password is first selected for a user, the password encryption program `passwd(1)` selects a random 12-bit number as the salt. The clear password string is then encrypted using this salt and the result is stored in the password file. Later on, when the user attempts to login to the system, the salt is extracted from the password file and is used to encrypt the user's typed password. The effect of salting is to allow for 4096 possible encryptions of the same password string.

Obviously, the use of salting does not necessarily improve the strength of the encryption. In fact, especially since the mechanism of DEA is not well understood by cryptanalysts who do not have access to classified files explaining the algorithm, it is possible that salting may have weakened the encryption process. However, the modification was done in order to prevent the use of hardware DES implementations in speeding up key searches, and also to prevent password cracking programs from precomputing commonly used passwords and storing them in a file or array and thus bypassing the (slow) encryption process.

On the surface, the UNIX `crypt()` function appears to have fulfilled all of its designers' aims. It is compact, appears at this stage to be irreversible, and software implementations of DEA tends to be slow, a password taking more than one second of CPU time to encrypt on a PDP-11/70.

However, there has been many doubts casted upon the strength of DES, including disagreement over whether a 56-bit key was sufficiently strong. Diffie & Hellman[Dif77a] predicted in 1977 that the DES algorithm could be compromised by a dedicated machine with around one million chips that can be built for around \$20 million. This machine could then search the complete key space in approximately one day. They also predicted that by 1990 hardware speeds would have improved so much that a 56-bit key would no longer be secure. The NCSC no longer certifies DES for even unclassified government information, a sure indication that DES is no longer considered secure. Furthermore, 25 applications of DEA does not necessarily improve the security of the basic algorithm, especially since the key schedule does not change between passes.

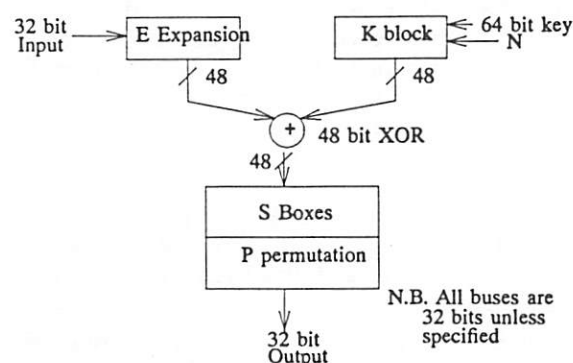
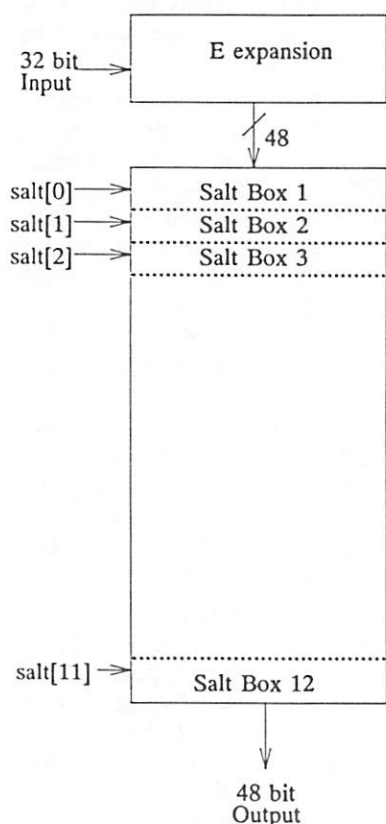
Most recently, Ali Shamir and Eli Biham[Sha89a] have reported that a chosen plaintext attack can reverse the DES encryption process in a time less than that required by exhaustive key search *provided* less than 16 rounds of the f function are run. It will be interesting to see if a variant of this method can be used for reversing the UNIX password encryption process, although this seems unlikely since the `crypt()` function uses 25×16 applications of the f function.

HARDWARE IMPLEMENTATION

In order to design hardware which will decrypt passwords in as short a time as possible, we must use components with a very small propagation delay. To this end, we chose the ECL 100K logic family.

f Function

The *f* function forms the heart of DEA and a password encryption involves 25 applications of DEA each of which make 16 applications of *f*. Figure 1 shows a block diagram of the *f* function.

Figure 1: The *f* functionFigure 2: The *E* Expansion*E* Expansion

The main difference between DEA and *crypt*(3) lies in the salting of the *E* expansion operation which outputs a 48-bit block from a 32-bit input. In DEA, this expansion is always performed in the same way, so we could implement this by rearranging the connecting wires. For *crypt*(3), the bits in the output of the *E* expansion may be exchanged according to the value of the salt.

The exchange of *E* output bits involves only optionally crossing two connections depending on whether a particular bit of the salt is set or cleared. Thus, the exchange can be implemented using 12 two-pole changeover relays, one for each bit of the salt. Each relay acts as a crossbar connection controlled by a bit from the salt, thus allowing the 32-bit input to pass through the normal DES *E* expansion and then through the salt-dependent permutation (see Figure 2). Since the salt need only be set once for each user, the speed of switching of the relays does not matter. Furthermore, during the encryption process, the signal only passes through the relay contacts, and so no propagation through logic gates is required. Hence the difference between the *E* expansion of DES and *crypt* does not affect the speed of this hardware encryption device. Figure 2 shows a block diagram of the *E* expansion and Figure 3 shows a blowup of a salt box.

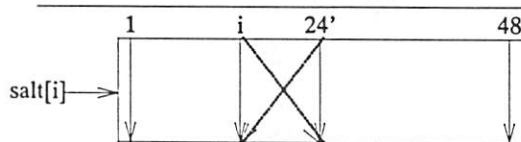


Figure 3: The Salt Box

Key Schedule

The key schedule converts *K* into a 48-bit block depending on the iteration number. Thus the subkey for the *i*-th iteration is *K_i* which is a selection of *K*. Our method of calculating the key schedule is generate all 16 key schedule values for each of the 48 bits of output, and then use a 100164 1-of-16 multiplexor to select the desired output for any given iteration (see Figure 4).

XOR of the E expansion with K_i

This 48-bit XOR is implemented using 10 100107 quint XOR gates which have a maximum propagation delay of 1.7 ns.

S selection and *P* permutation

The XOR described above is passed through the *S* selection boxes and then permuted. Note that the *S* boxes are arranged as 8 groups of selections of 4 bits from 6 bits, and so 8 64×4 bit ECL RAMS are required. We use 100422's which have 5 ns

propagation delay from the address input to the output. The output is then permuted according to the P permutation which just involves crossing of wires.

Final XOR

To complete the f function we do the 32-bit XOR of the output of the above permutation with the leftmost 32 bits of the previous iteration.

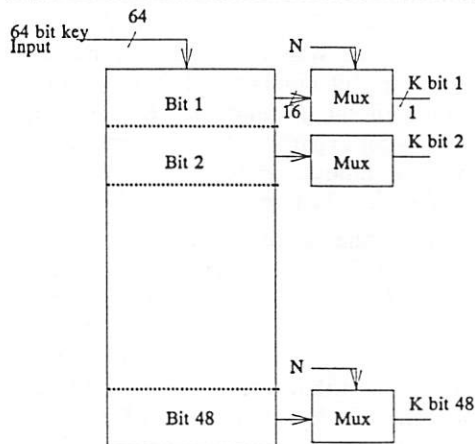


Figure 4: The Key Schedule

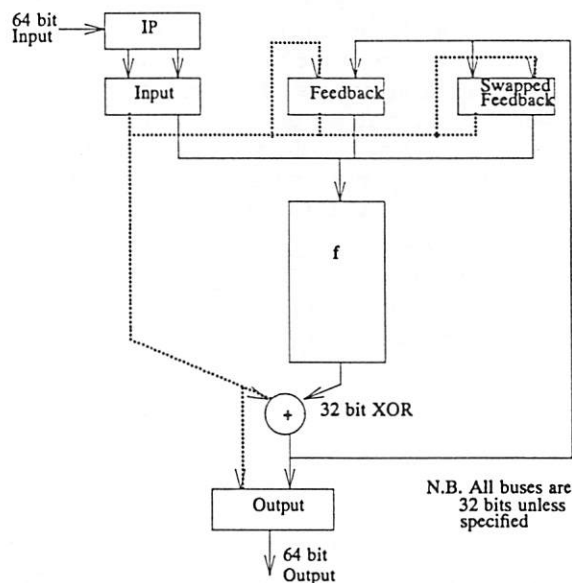


Figure 5: Block Diagram of the DES Hardware

Block Transformation

Block transformation involves the exchange of the leftmost 32 bits of the 64-bit word with the rightmost 32 bits. As shown in Figure 5, we have three latches that can feed the f function, and this allow us to optionally perform block transformation or clear the 64-bit input to f . Clearing is required at the start of the *crypt* operation. The three latches are wire-ORed together and the two latches not being used are cleared. The three 64-bit latches require 33 100151 hex flip-flops.

State Machine

A finite state machine controls the flow of information through the circuit. It must feed the key schedule computation unit with the correct iteration number, select the correct input to the f function from the three latches and latch the output result when the computation has been completed. Figure 5 shows a block diagram of the hardware.

Note that DES can be implemented using the same hardware by setting the E expansion relays to flow straight through. Then the main difference between *crypt* and DES is that *crypt* performs 25 iterations of the DES algorithm.

Operating Frequency

The gate delays for a single iteration of the algorithm are the delays for 2 XORs, 1 multiplexor, 1 RAM and one latch which totals to a worse case of 14.7 ns. We use a cycle period of 15 ns which corresponds to a frequency of 66 MHz. Hence it takes 6 μ s to encrypt a password.

As a DES encryption machine, the board can process 64 bits every 240 ns, that is, at a rate of 267 Mbps. It is interesting to note that a recently reported single chip implementation of DES[Ver88a] operates at 20 Mbps.⁴

SOFTWARE IMPLEMENTATION

Although a hardware implementation of *crypt*() is within range of a determined cracker, we also decided to implement a fast software version. This implementation is substantially faster than the UNIX routine and is portable across any hardware platform with native 32-bit operations. Interestingly, our implementation does not discriminate against either big endian or little endian machines, although as currently implemented, it seems to shine on RISC (Reduced Instruction Set Computer) architectures due to the fact that the implementation does not tend to require complex instruction addressing modes but requires a fast basic instruction execution cycle, two characteristics which have been used to describe RISC architectures. Our implementation was written in Australia and is hence free of any US export restrictions.

A good description of the Internet Worm/Virus implementation of *crypt*() is given in Seeley[See89a] and we used this implementation as a base for our own approach. Our initial implementation encrypted a password on the Sun Sparcstation 1 in just over 6 ms. The performance of this implementation was disappointing compared to Bishop[Bis88a] so we reimplemented the program using these new ideas.

⁴This is perhaps an unfair comparison as the reported chip implements far more than just the encryption part of DES.

This resulted in an implementation that encrypts a password on the same machine in just over 2 ms. The following notes describe our second implementation.

The basic speedup over the UNIX implementation was due to bit compaction into machine words. The UNIX implementation uses one byte to store every bit that needs to be manipulated. Hence, 64 bytes consisting of the numbers 0 and 1 were used to represent a 64-bit entity. In our implementation, the same entity is represented by two 32-bit words. This allows us to use the rotate and exclusive OR operations in the instruction set, and hence exploit the inherent parallelism in the datapath of the CPU. Also, we precomputed all expansion, selection, and permutation functions and in many cases combined several operations into one precomputed array.

As an example, the $PC1$, LS_i , $PC2$ operations can be effectively combined into a single operation which we call $keys_i$. This operation can then be precomputed so that instead of performing the operation on every bit in the 56-bit key yielding a 48-bit subkey, we can divide the original 56-bit key into eight groups of 7-bit blocks. Each 7-bit block is used to index into an array of precomputed 48-bit blocks. The eight resultant 48-bit blocks are then ORed together to form the subkey. As an example, we declare and precompute an array used for key schedule computations called $keys$, the DES key K is in the array k and the computed key schedules will be stored in the array $keysched$:

```
typedef unsigned long Word;
typedef unsigned char Byte;

static Word keys[16][8][128][2];
static Byte k[8];
static Word keysched[16][2];
```

Note that two Words are used to store the 48-bit quantity, which is divided into 24-bit halves, each of which fits in the 32-bit machine word. For example, to compute the i th subkey, all we need to do is to use each byte in k to index into the $keys$ array and then OR the results into the $keysched$ array.

```
keysched[i][0] = keys[i][0][k[0]][0] |
                 keys[i][1][k[1]][0] |
                 keys[i][2][k[2]][0] |
                 keys[i][3][k[3]][0] |
                 keys[i][4][k[4]][0] |
                 keys[i][5][k[5]][0] |
                 keys[i][6][k[6]][0] |
                 keys[i][7][k[7]][0];
keysched[i][1] = keys[i][0][k[0]][1] |
                 keys[i][1][k[1]][1] |
                 keys[i][2][k[2]][1] |
                 keys[i][3][k[3]][1] |
                 keys[i][4][k[4]][1] |
                 keys[i][5][k[5]][1] |
                 keys[i][6][k[6]][1] |
                 keys[i][7][k[7]][1];
```

Note that all arrays are declared as static variables rather than left on the stack so that the compiler can generate actual memory references or memory plus

register offset references rather than indirect stack references. This significantly speeds up code execution on CISC (Complex Instruction Set Computer) machines. The subkey computation loop was also unrolled to simplify the compiler address generation.

A similar technique is used to perform the f function and the FP final permutation.⁵ In the f function, we note that f accepts a 32-bit argument, which then immediately expands to a 48-bit block through the E expansion. The result of the f function is a 32-bit number which is then exclusively ORed with another 32-bit number and then fed into the next invocation of the f function. First of all, note that since E is an expansion which maps every 32-bit block into a unique 48-bit block, we can obtain the inverse of E which we shall call E^{-1} .

Suppose we define a function g such that $g(X,K)=E(f(E^{-1}(X),K))$ then we notice that $g(X,K)=E(P(S(X \oplus K)))$. In other words, we can combine the E , P and S operations into a single operation that can be precomputed. We can then transform the DEA algorithm into 16 applications of the g function followed by application of E^{-1} on both halves which is then fed into the FP final permutation.

Bishop[Bis88a] gives a full mathematical treatment of the modified algorithm outlined above. Finally, we note that the effect of salting can be obtained by exchanging bits of the result of the E expansion. Given that we are representing a DES block as two machine words, we can calculate the salted expansion by performing several exclusive-ORs and one bitwise AND ($\&$) operation.

$$\begin{aligned} \text{Let } UD &= E(X) \\ &= u_1 u_2 \cdots u_{24} d_1 d_2 \cdots d_{24} \\ E'(X) &= (U \oplus M)(D \oplus M) \\ \text{where } M &= (U \oplus D) \& \Psi' \\ \Psi' &= s_1 s_2 \cdots s_{12} 0 \cdots 0 \end{aligned}$$

Table 4: Salting the E expansion

In our actual implementation, we reinitialize the precomputed EPS array given a new salt in order to save the time required to perturb the result of the E expansion. This is because in a password cracking situation, the cost of precomputation whenever the salt changes is insignificant as many password guesses are made for every encrypted password.

⁵The IP permutation is not necessary since the text that is encrypted is always the zero block and we observe that $0=IP(0)$. Also, since $FP(IP(X))=X$, we never have to perform FP until after the 25 iterations of the DEA as the results of each iteration is fed directly into the next iteration.

GUESSING PASSWORDS

We shall not describe the design and implementation of the Internet Worm/Virus's password cracking routines because it has already been documented in existing literature,[Eic89a,See89a,Spa88a] although the methods it uses can be generalized for any password guessing program. Essentially, a password guessing program works by reading in the password file and then making multiple guesses of the password of every user or selected users in the password file. The selection of password guesses is vitally important as the better the quality of the guesses, the greater the chance of actually hitting the correct password within a given period of time.

Each password guess has to be encrypted using the current salt and then matched against the encrypted password. Obviously, the effectiveness of this procedure strongly depends on how fast a password can be encrypted. Ideally, the encryption process should take almost no time so that a complete key search can be done. Since we know that the encryption process takes up a significant amount of time, even with hardware assistance, it is more effective to implement a good password guessing generator and use brute force search only as a last resort.

A password guesser should make intelligent password guesses which is dependent on the personality traits and characteristics of the person who has chosen the password. Ideally, personal information concerning the password creator should be known to the program, such as names and birth dates of people, car registration numbers etc. In practise, this information is very hard to obtain, but a good start can be made by scanning the password file itself for information about users. The password file often stores very useful pieces of information which can be used, such as the user's full name, his/her phone extension and/or office number. A password guesser should certainly try permutations of the user's login name, full name and any other detail known about the user. Searches through lists of words or combination of words can also be effective. These may include lists of first and last names, words occurring in a special context (swear words, technological jargon, biblical and mythological names), or even dictionaries.

If a brute force search is attempted, this can be made more efficient by ordering the keys searched so that common characters and sequences of characters appear first in the search. Any additional information such as the first character of the password or even which side of the keyboard it was typed on, will dramatically reduce the time required to decrypt a password.

Table 5 shows the speed of our software implementation of *crypt* on a wide variety of machines. Figure 6 summarizes the results of this table in a

scatter plot.

Table 6 demonstrates the speed difference between hardware and software implementations and also shows the time required to decode a password using a brute force search. The software times were extrapolated from the RS/6000 encryption time. It is easily seen that for lower case alphabetic characters (which form the majority of passwords), it is very feasible on our hardware. The table also shows that such brute force searches are not possible in software on commonly available workstation class computers.

Machine name	CC	Time (ms)
Sun 3/50	cc -O4	16.0
Sun 3/60	cc -O4	9.5
Sun 3/60	gcc -O	7.2
Pyramid 9810	gcc -O	6.3
DECstation 2100	cc -O	3.3
Sparcstation 1+	cc -O4	2.0
DECsystem 5000/200	cc -O	1.8
IBM RS/6000-530	cc -O	1.2
ECL hardware	n/a	0.0060

Table 5: *crypt*() speed Comparison

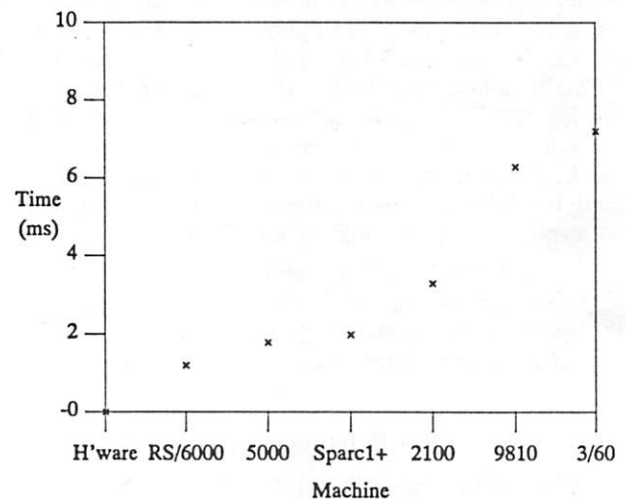


Figure 6: Scatter plot of *crypt*() performance

Search Criterion	Number of Passwords	H'ware (days)	S'ware (days)
Lower case only	26 ⁸	14.5	3712
As above + digits	36 ⁸	196	39182
All alphabetic	52 ⁸	3712	742496

Table 6: Brute force search times

It is interesting to note that the original UNIX password encryption algorithm based on the M-209 cipher machine was changed because it could be implemented on a PDP-11/70 in 1.25 ms and this was deemed to be too fast.[Mor78a] Since even our software version can perform a password encryption in less time than this, it may be time to change the

current method of encryption yet again.

There are a number of possible improvements to password encryption algorithm that will significantly decrease the success ratio of a password encryption program that uses our hardware or software implementations of the *crypt()* function. An easy method would be to use the next eight characters of the password as the initial input to the DEA and then modifying the *passwd(1)* program to only allow passwords longer than eight characters. Alternatively, a concept similar to the *shadow password file* idea can be implemented by UNIX administrators to stop users and/or programs from reading the password file.

These implementations are by no means the last word on speedy password encryption. It is certainly tempting to imagine the speedup that can be obtained using a massively parallel computer such as the Connection Machine [Hil85a] or through the use of a large array of custom VLSI chips which can test passwords in parallel.

CONCLUSION

A design of a very fast hardware encryption device has been presented in this paper. Such a device makes brute force searching of passwords possible due to the small key space from which people normally select passwords. It was shown that perturbing the *E* expansion of the DES algorithm with the salt does not result in any change in the speed of the implementation of *crypt()* in hardware, although applying DES 25 times reduces the speed at which we can encrypt passwords by a factor of 25.

A software implementation of the UNIX password encryption algorithm was also presented, and the speed of this implementation was compared with that of the custom hardware.

References

- Duf89a. Tom Duff, "Viral Attacks on UNIX System Security," *USENIX Winter '89 Conference Proceedings*, (1989).
- Rit78a. Dennis Ritchie, "On the Security of UNIX," in *UNIX Programmers Manual*, (3 April 1978).
- Gra84a. F. T. Grampp and R. H. Morris, "UNIX Operating System Security," *AT&T Bell Laboratories Technical Journal* 63(8 (Part 2))(October 1984).
- Bac86a. Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice Hall International, Inc. (1986).
- Eic89a. Mark W. Eichin and Jon A. Rochlis, "With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988," *IEEE Symposium on Research in Security and Privacy*, (9 February 1989).
- Sec89a. Donn Seeley, "A Tour of the Worm," *USENIX Winter '89 Conference Proceedings*, (1989).
- Spa88a. Eugene H. Spafford, "The Internet Worm Program: An Analysis," *Purdue Technical Report*, (CSD-TR-823)(28 November 1988).
- Mor78a. Robert Morris and Ken Thompson, "Password Security: A Case History," in *UNIX Programmers Manual*, (3 April 1978).
- Ano77a. Anon, "Data Encryption Standard," *FIPS PUB*, (46)National Bureau of Standards, (15 January 1977).
- FIP75a. FIPS, *Proposed Federal Information Processing Data Encryption Standard*, Federal Register (17 March 1975).
- Seb89a. Jennifer Seberry and Josef Pieprzyk, *Cryptography: An Introduction to Computer Security*, Prentice Hall Australia (1989).
- ANS81a. ANSI, *American National Standards Data Encryption Algorithm*, American National Standards Association (1981).
- Dif77a. W. Diffie and M. E. Hellman, "Exhaustive Cryptanalysis of the NBS Data Encryption Standard," *Computer* 10 pp. pp. 74-84 (June 1977).
- Sha89a. Ali Shamir and Eli Biham, "Differential Cryptanalysis of DES-like Cryptosystems," *Crypto '89*, (1989).
- Ver88a. Ingrid Verbauwhede, Frank Hoornaert, Joos Vandewalle, and Hugo de Man, "Security and Performance Optimization of a New DES Data Encryption Chip," *IEEE Journal of Solid State Circuits* 23(3) pp. 647-656 (June 1988).
- Bis88a. Matt Bishop, "An Application of a Fast Data Encryption Standard Implementation," *Dartmouth College Technical Report*, (PCS-TR88-138)Department of Mathematics and Computer Science, (1988).
- Hil85a. W. Daniel Hillis, *The Connection Machine*, MIT Press (1985).
- Philip Leong works at the Systems Engineering and Design Automation Laboratory at the Department of Electrical Engineering at the University of Sydney. His interests include operating systems, digital signal processing and VLSI design. He received a B.Sc. degree in Computer Science in 1987 and a B.E. (Hons) in Electrical Engineering in 1989. Reach him via mail at Systems Engineering and Design Automation Laboratory; Department of Electrical Engineering;



University of Sydney J03; NSW 2006 AUSTRALIA;
Phone: +61 2 692-3297 His electronic mail address
is phwl@ee.su.oz.au.

Chris Tham is currently employed as a Treasury Analyst at the State Bank of Victoria. His interests include distributed operating systems, concurrent computer language design and computer music. He graduated from University of Sydney in 1988 with a B.Sc. (Hons) in Computer Science. Reach him via mail at State Bank of Victoria; Level 23 9 Castlereagh St; Sydney NSW 2000; AUSTRALIA; Phone: +61 2 239-6282. His electronic mail address is christie@blueboy.ct.saleven.oz.au



Appendix A

```

/*
 * UNIX compatible version of crypt(3)
 * that uses fast DES routines
 */

#ifndef TRACE
#include <stdio.h>
#endif
#include "des.h"
#include "efp.h"
#include "spe.h"
#include "keys.h"

#define f(left, right, i) \
{ \
    register Word    ss; \
    TRACEOUT("right", writeBlock48(&right)); \
    t1.w[0] = (right).w[0] ^ keysched[--i].w[0]; \
    t1.w[1] = (right).w[1] ^ keysched[i].w[1]; \
    TRACEOUT("key", writeBlock48(&keysched[i])); \
    t2.w[0] = \
        spe[0][t1.h[0]][0] | \
        spe[1][t1.h[1]][0] | \
        spe[2][t1.h[2]][0] | \
        spe[3][t1.h[3]][0]; \
    t2.w[1] = \
        spe[0][t1.h[0]][1] | \
        spe[1][t1.h[1]][1] | \
        spe[2][t1.h[2]][1] | \
        spe[3][t1.h[3]][1]; \
    TRACEOUT("t2", writeBlock48(&t2)); \
    ss = (t2.w[0] ^ t2.w[1]) & m; \
    (left).w[0] ^= t2.w[0] ^ ss; \
    (left).w[1] ^= t2.w[1] ^ ss; \
    TRACEOUT("left", writeBlock48(&left)); \
}

static Block64 keysched[16];
static Block64 left, right;
union result
{
    Byte    b[9];
    Word    w[2];
} block;
static char iobuf[16];
static Word m;

void
setsalt(salt)
char *salt;
{
    register int    i, j;

    m = 0;
    for (i = 0; i < 2; i++)
    {
        char    c;

        iobuf[i] = c = *salt++;
        if (c > 'Z') c -= 6;
        if (c > '9') c -= 7;
        c -= '.';
        for (j = 0; j < 6; j++, c >>= 1)
        {
            m <<= 1;
            if (c & 1)
                m |= 1;
        }
    }
}

#ifdef LITTLE_ENDIAN
    m <<= 16;
#endif

char *
encrypt(pw)
char *pw;
{
    register int    i, j;

```

```

Word    s1, s2;

/*
 * keysched is stored in reverse
 * order to keys for optimization
 */
memset(keysched, 0, sizeof(keysched));
for (i = 0; (j = *pw++) && i < 8; i++)
{
    keysched[15].w[0] = keys[0][i][j][0];
    keysched[14].w[0] = keys[1][i][j][0];
    keysched[13].w[0] = keys[2][i][j][0];
    keysched[12].w[0] = keys[3][i][j][0];
    keysched[11].w[0] = keys[4][i][j][0];
    keysched[10].w[0] = keys[5][i][j][0];
    keysched[9].w[0] = keys[6][i][j][0];
    keysched[8].w[0] = keys[7][i][j][0];
    keysched[7].w[0] = keys[8][i][j][0];
    keysched[6].w[0] = keys[9][i][j][0];
    keysched[5].w[0] = keys[10][i][j][0];
    keysched[4].w[0] = keys[11][i][j][0];
    keysched[3].w[0] = keys[12][i][j][0];
    keysched[2].w[0] = keys[13][i][j][0];
    keysched[1].w[0] = keys[14][i][j][0];
    keysched[0].w[0] = keys[15][i][j][0];
    keysched[15].w[1] = keys[0][i][j][1];
    keysched[14].w[1] = keys[1][i][j][1];
    keysched[13].w[1] = keys[2][i][j][1];
    keysched[12].w[1] = keys[3][i][j][1];
    keysched[11].w[1] = keys[4][i][j][1];
    keysched[10].w[1] = keys[5][i][j][1];
    keysched[9].w[1] = keys[6][i][j][1];
    keysched[8].w[1] = keys[7][i][j][1];
    keysched[7].w[1] = keys[8][i][j][1];
    keysched[6].w[1] = keys[9][i][j][1];
    keysched[5].w[1] = keys[10][i][j][1];
    keysched[4].w[1] = keys[11][i][j][1];
    keysched[3].w[1] = keys[12][i][j][1];
    keysched[2].w[1] = keys[13][i][j][1];
    keysched[1].w[1] = keys[14][i][j][1];
    keysched[0].w[1] = keys[15][i][j][1];
}

/* clear working blocks */
left.w[0] = 0;
left.w[1] = 0;
right.w[0] = 0;
right.w[1] = 0;

j = 12;
goto middle;
while (j--)
{
    static Block64 t1, t2;

    /*
     * Do 16 rounds of the f() function
     * on even rounds the right half is
     * fed to f() and exclusive ored with
     * the left half and vice versa
     */
    for (i = 16; i;)
    {
        f(right, left, i);
        f(left, right, i);
    }

middle:
    for (i = 16; i;)
    {
        f(left, right, i);
        f(right, left, i);
    }
}

s1 = (left.w[0] & -m) | (left.w[1] & m);
s2 = (left.w[1] & -m) | (left.w[0] & m);
left.w[0] = s1;
left.w[1] = s2;
s1 = (right.w[0] & -m) | (right.w[1] & m);
s2 = (right.w[1] & -m) | (right.w[0] & m);
right.w[0] = s1;

```

```

right.w[1] = s2;
block.w[0] =
    efp[0][right.h[0]][0] |
    efp[1][right.h[1]][0] |
    efp[2][right.h[2]][0] |
    efp[3][right.h[3]][0] |
    efp[4][left.h[0]][0] |
    efp[5][left.h[1]][0] |
    efp[6][left.h[2]][0] |
    efp[7][left.h[3]][0];
block.w[1] =
    efp[0][right.h[0]][1] |
    efp[1][right.h[1]][1] |
    efp[2][right.h[2]][1] |
    efp[3][right.h[3]][1] |
    efp[4][left.h[0]][1] |
    efp[5][left.h[1]][1] |
    efp[6][left.h[2]][1] |
    efp[7][left.h[3]][1];
block.b[8] = 0;
TRACEOUT("block", writeBlock64(&block));

for (i = 0; i < 11; i++)
{
    int    type = i * 6;
    int    pos = type >> 3;
    char    c;

    switch (type & 07)
    {
        case 0:
            c = block.b[pos] >> 2;
            break;

        case 2:
            c = block.b[pos] & 077;
            break;

        case 4:
            c = ((block.b[pos] & 0xf) << 2)
                + (block.b[pos + 1] >> 6);
            break;

        case 6:
            c = ((block.b[pos] & 03) << 4)
                + (block.b[pos + 1] >> 4);
            break;
    }

    c += '.';
    if (c > '9') c += 7;
    if (c > 'Z') c += 6;
    iobuf[i + 2] = c;
}
iobuf[i + 2] = 0;

return iobuf;
}

char *
crypt(pw, salt)
char *pw, *salt;
{
    setsalt(salt);
    return encrypt(pw);
}

```


An Authentication Mechanism for USENET

Matt Bishop – Dartmouth College

ABSTRACT

As UNIX based systems become more ubiquitous, so does the international news network and bulletin board system USENET. Like electronic mail, the USENET has no security whatsoever; forging articles, or altering posted articles in transit, is trivial. In the past, this has not been a problem, but with the advent of “authoritative” news groups such as `comp.bugs.4bsd-fixes` (which contain “official” bug fixes and enhancements from Berkeley), the integrity and authenticity of some postings becomes paramount.

The Privacy and Security Research Group, working under the auspices of the Internet Research Steering Group, recently released a set of proposals to enhance the security of electronic mail. One proposal adds to the existing mail handling structure by adding an extra layer of (security) processing between the transport and user agents; the second describes a certificate-based key distribution and management infrastructure for public key cryptosystems that supports the first.

This paper discusses the design of an addition to network news based upon the security enhancements being added to electronic mail. It uses the same underlying key distribution and management infrastructure, so it does not require new key management protocols or software, but merely the integration of existing protocols. Further, it is completely compatible with unauthenticated news, so it need not be adopted wholesale, but can be employed on a site-by-site basis. We also will discuss expected efficiency of the system.

We also contrast this scheme with some others such as the existing *nntp* authentication scheme and with the use of Kerberos. Advantages and disadvantages of the schema will be described.

Authenticity, Integrity, and the News

The bulletin board system named USENET has become ubiquitous in the UNIX world. Messages of all varieties are posted to it, and greeted with varying degrees of belief. Many are inconsequential (a false message of the form “1977 Toyota for sale” is hardly catastrophic); but some are not, and even inconsequential messages can create quite a bit of confusion.

Perhaps the most amazing, and amusing, hoax involving a forged USENET article is the now-infamous *kremvax* posting perpetrated as an April Fools’ joke [BEER84]; unfortunately, a good many people failed to realize this, and (apparently) thought that Chernenko¹ had actually posted something to the USENET. When Beertema revealed the hoax two weeks later, many people (including several who did not realize this was a hoax) were furious at the deception.

More serious was the “letter bomb” incident a few years before that. One of the more pleasant aspects of USENET is that it encourages the exchange of software, which is usually packaged

into an archive called a *shar* file. To unpack such an archive, the user simply saves the article (minus any headers) in a file and sends the file to a shell as standard input. The archive contains shell commands that reconstruct the sources. A malicious poster circulated such an archive; among the shell commands, it contained

```
cd $HOME; rm -rf *
```

Anyone who did not check the source carefully lost the files in their directory.

Ordinarily, one could simply say “don’t trust anything you see on USENET unless it is independently confirmed.” The difficulty with such a statement is that the medium is quite useful for propagating important information quickly; for example, the Computer Science Research Group at the University of California at Berkeley uses the USENET group `comp.bugs.ucb-fixes` to spread word of important bug fixes (usually related to security). Should one of these postings be ignored, the system administrator may be leaving a security vulnerability in place; however, should a posting to this group be tampered with, or forged, and *then* installed, one could unwittingly introduce a very serious security hole. By the time Berkeley could spread word of the forgery (or changed article), the attacker could

¹Then the General Secretary (head) of the Communist Party of the Union of Socialist Soviet Republics.

have wreaked quite a bit of havoc.

News administrators deal with a very minor variant of this quite often. Among the many newsgroups is the *control* group, through which newsgroups can be added and deleted automatically. Usually the first step of configuring the news package is to disable the automatic processing of control messages that create and delete newsgroups, because of the large number of forgeries that occur.

The USENET is an example of a *message system* [X50087]; so is electronic mail. Indeed, the USENET resembles a very large electronic mailing system, where readers are recipients of the "letters" (articles); the only difference is that articles are stored in a single system "mailbox," whereas letters are stored in mailboxes associated with each user. In general, message systems may be thought of as *user agents*, or components which interact with users, and *message transport agents*, which pass messages from one system to another. Figure 1 illustrates the relationship of these components. For electronic mail, the user agents are the various mail sending and reading programs, and the message transport agents are the *uucp* and *rmail* programs or implementations of SMTP [POST82]. For news, the user agents are the various news reading and posting programs, and the message transport agents are the *nntp*, *uucp*, and *rnews* programs.

The goal of this paper is to present a proposal which enables electronic news articles to be *authenticated* to confirm the (claimed) identity of the poster, and *integrity checked* to confirm that the article has not been changed since it was posted. it was posted. A second goal is to restrict all necessary changes to user agents, and not to the message transport agents to be changed at all. This also implies that any changes must be implemented compatibly with existing news systems. Given that control of the USENET is completely decentralized, no authority could require all sites to use one particular version of news programs; and even if sites decided to

adopt a version of the news programs offering authentication and integrity, being incompatible with the software at other sites would fragment the USENET.

The incentives for such a mechanism are powerful. Given authenticated, integrity-checked articles, there would be a considerable disincentive to post malicious logic (such as the letter bomb above), as the origin could be proved or the tampering detected. This paper discusses one mechanism for doing this. The next section looks at a set of Internet draft standards that provide both a mechanism and an infrastructure for private, integrity-checked, and authenticated electronic mail; we then consider adopting a similar mechanism and using that infrastructure. We estimate the performance of the resulting mechanism, and conclude by comparing and contrasting it with several other possible methods.

Authenticity, Integrity, and Mail

If authentication and integrity can be made available for electronic mail, then the same mechanism should be able to provide authentication and integrity for electronic news as well. Using the same mechanism would also allow USENET to benefit from an infrastructure which can be shared with other services, rather than having to define its own unique arrangement.

The issue of providing authentication and integrity in Internet electronic mail was first seriously discussed in [THOM74]. Recently, the Privacy and Security Research Group has circulated a set of draft Internet standards [KENT89], [LINN89a], [LINN89b] proposing mechanisms and an organizational structure to provide these features, as well as privacy, for Internet mail [BISH90]. This mechanism is completely upwards-compatible with existing mail conventions, and requires only that user agents be modified; no changes to the transport mechanisms are required. While privacy is (in

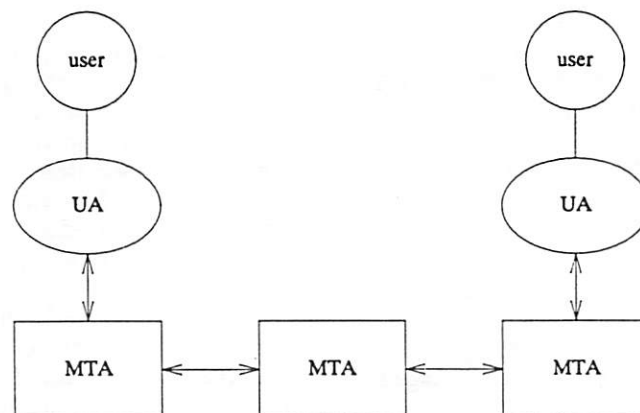


Figure 1: The Message Handling System Model

general) not relevant to USENET, both integrity and authenticity are, and the mechanisms and infrastructure of privacy-enhanced mail (as the set of proposals is collectively called) can be used for USENET.

Like other network communications schemes involving cryptography [VOYD83], privacy-enhanced electronic mail uses two keys: the first is a per-letter *data exchange key* generated pseudorandomly, and the second is an *interchange key* associated with a user (or with a pair of users) and changed (relatively) infrequently. To check authenticity and integrity, a checksum is generated from the message (possibly using the data exchange key) and that checksum is encrypted using the interchange key. In more detail:

- (1) The message is *canonicalized*; that is, all characters are converted to their ASCII equivalents, and line delimiters are changed to <CR><LF>. This ensures that any computations made based on the contents of the message can be duplicated on any system.
- (2) An integrity check is then computed (possibly using the data encryption key) and enciphered using the interchange key. The draft standards specify acceptable algorithms for this integrity checksumming. The additional data (keys, algorithms used, and checksums) are placed in the body of the message between the SMTP headers and the text, as shown in Figure 2. To the message transport agents, these new headers are simply part of the body of the message, and hence are ignored; but user agents appropriately modified will treat these lines as being special.

RFC-822 (standard SMTP) headers go here
 <blank line>
 -----PRIVACY-ENHANCED MESSAGE BOUNDARY-----
 RFC-1113 (privacy enhanced mail) headers go here
 <blank line>
 text of message
 -----PRIVACY-ENHANCED MESSAGE BOUNDARY-----

Figure 2: Encapsulation of Security Data

Interchange keys can be based either on classical cryptosystems, which involve a secret key known to both the sender and recipient only, or public-key cryptosystems, which involve a public key known to everyone (including the recipient) and a private key (known only to the sender). Both systems can be used for authentication [DENN82], and privacy-enhanced electronic mail supports interchange keys used in either type of cryptosystem. However, keys for classical cryptosystems are assigned to *pairs* of users (namely, the sender and the recipient). This is unsuitable for a medium such as USENET, where communications are broadcast from one user (the poster) to all others.

Managing interchange keys for a public-key cryptosystem can most conveniently be done using *certificates*. A certificate contains a user's identification, his/her public key, the name of the issuer of the certificate, some other certificate information, and a checksum binding all the above together. The checksum, of course, is encrypted using the issuer's private key; and associated with each issuer is a certificate containing the issuer's public key. Thus, these certificates form a *certification hierarchy* (see Figure 3) and enable a user to validate the certificate of a sender easily:

- (1) The user obtains the certificate of the sender, possibly from an Internet directory server, or possibly from the letter's headers.
- (2) The user then obtains the certificate of the issuer and extracts the issuer's public key.
- (3) The user decrypts the checksum on the certificate being validated, and recomputes the checksum and compares. If they differ, either the issuer's certificate is bogus, or the sender's certificate is bogus.
- (4) If the user wishes to validate the issuer's certificate, he/she can simply repeat steps (2) and (3) until the root of the hierarchy is reached. The user knows the public key of the root by out-of-bands methods (such as being sent that

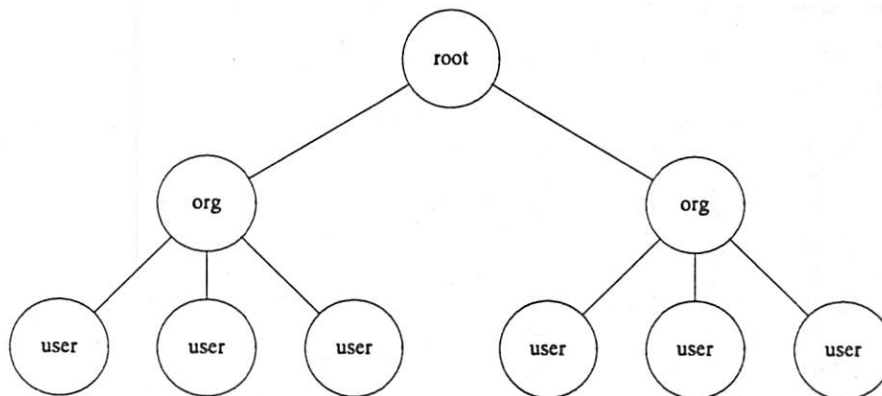


Figure 3: Sample Certification Hierarchy

information when he/she registers a certificate with the root) and can validate that certificate separately.

One problem with certificates is that they are not composed of only printable characters because the keys are represented as bit sequences. But most SMTP implementations do not support transmitting arbitrary bit sequences, so encoding the certificates is necessary to enable them to be transmitted through mail. The certificate is broken up into sets of six bits, and each set is mapped to a printable character (see Figure 4).

An example of an infrastructure supporting this arrangement is presented at length in [KENT89] for certificates based on the RSA cryptosystem [RIVE78]. The important point is that the privacy enhanced mail protocols separate the key distribution scheme from the message encoding scheme. This way, users operating in an environment where they could trust a central server to manage interchange could do so, whereas users in an environment without such a server could use a certificate-based key distribution mechanism. The latter point is vital to the USENET solution, since it allows public keys to be widely distributed and yet be trustworthy without requiring trusted key servers.

A Proposed Solution for USENET

First, note that privacy is irrelevant in the context of a news system (like USENET), so we need only worry about integrity and authenticity. The proposal for USENET parallels those parts of the solution for electronic mail involving mechanisms to ensure integrity and authenticity. First, we discuss the modifications to the relevant standards.

The format for message interchange on USENET is defined by [HORT87]. We propose adding a second set of message headers; however, as the current news interchange standard *explicitly* requires that unknown header fields be passed through untouched, we can omit the encapsulation

step in privacy-enhanced electronic mail and simply add the new header fields in the header without having to change any of the transport agents. (In other words, the encapsulation mechanism that privacy enhanced electronic mail requires to ensure compatibility with existing message transport agents is completely unnecessary for USENET.) As for user agents, news readers supporting authentication and integrity checking would be modified to use the data in these new headers; unmodified news readers would simply ignore them.

The first identifies the sender and his or her certificate:

Sender-ID: sender:issuer:cert_id

The **sender** is the sender's unique name and assumes the form *user@host*, where *user* is unique to *host* and *host* is unique to the USENET. As UUCP sites need not have unique names, this suggests strongly that *host* should be a fully qualified domain name whenever possible. The **issuer** is the name of the authority that issued the certificate to *sender*, and **cert_id** is simply the serial number of that certificate.

The second header field is used only when the integrity checksum algorithm requires a data encryption key:

Key-Info: ik_use,dek

The **ik_use** is the name of the algorithm used to encrypt the data encryption key; **dek** is the data encryption key, encrypted using the sender's private key². Hence the data encryption key can be obtained by decrypting the second field (using the cryptosystem named in the first field) using the public key of the sender.

The third header field identifies how the message integrity check was computed and encrypted, and what that result is:

MIC-Info: alg,ik_use,mic

Here, **alg** is the algorithm used to compute the message integrity check, **ik_use** is the

¹These characters were chosen because they are not special to any message transport agent implementing SMTP (or the news protocols, for that matter) when they appear in the body of the message.

²This is opposite to what privacy-enhanced mail does; it encrypts the data encryption key using the *recipient's* public key.

0 A	8 I	16 Q	24 Y	32 g	40 o	48 w	56 4
1 B	9 j	17 R	25 Z	33 h	41 p	49 x	57 5
2 C	10 K	18 S	26 a	34 i	42 q	50 y	58 6
3 D	11 L	19 T	27 b	35 j	43 r	51 z	59 7
4 E	12 M	20 U	28 c	36 k	44 s	52 0	60 8
5 F	13 N	21 V	29 d	37 l	45 t	53 1	61 9
6 G	14 O	22 W	30 e	38 m	46 u	54 2	62 +
7 H	15 P	23 X	31 f	39 n	47 v	55 3	63 /

Figure 4: Printable encoding

cryptographic algorithm used to encipher the resulting MIC, and mic is the enciphered MIC. Note that the decryption key associated with the MIC is in the certificate named in the `Sender-Id` line, and if a cryptographic key were required, that key would be found on a `Key-Info` line.

The fourth and fifth header fields are included simply for convenience, but as many USENET sites will not have access to network-based directory servers, they will prove quite useful. The header field

`Certificate: send_cert`

contains the poster's certificate `send_cert`, encoded as described above, and

`Issuer-Certificate: issuer_cert`

gives the (encoded) certificate of the issuer of the certificate in the `Certificate` or `Issuer-Certificate` line immediately preceding. There may be any number of the latter lines, allowing the recipient to validate certificates up to some known issuing authority.

To authenticate a message, and verify it has not been changed in transit, the news reader would have to do the following:

- (1) Extract the sender's identification from the letter and obtain the corresponding certificate. This may require validation of that certificate.
- (2) If present, extract the data exchange key.
- (3) Extract and decrypt the given integrity checksum.
- (4) Canonicalize the message and recompute the integrity checksum.
- (5) Compare the computed with the transmitted integrity checksum. If they match, the message was indeed posted by the sender and has not been altered in transit. If they do not match, either the message is a forgery or it has been altered in transit.

From the software point of view, this work requires two modules. One module interacts with the posting software: given an article to be posted, it computes a checksum, encrypts that using the poster's private key, and inserts the appropriate header fields into the article's header. A second module interacts with the news readers: given an article to be checked, it retrieves the certificate, and authenticates and integrity checks the article. Note that these two modules can simply take as input an article, and return a code for "bogus" or "authentic and unaltered." Hence they do not require any major changes to news readers or posting programs, and indeed those programs need not always invoke the authentication routines. Thus, authenticated and unauthenticated news messages can coexist; this is

Estimates of Efficiency

As this authentication mechanism has not yet been implemented in news programs, what follows are simply estimates of efficiency. Ignoring the overhead of parsing the headers, the two greatest penalties will come from the computation of the checksum and its encryption. One algorithm to compute the integrity checksum is the cipher block chaining mode of the Data Encryption Standard [FIPS80]. Fast software and hardware implementations of the DES are available [BISH88]. Figure 5 lists some specific newsgroups³ where authenticity and integrity would be vital, shows the average number of bytes in those newsgroups' articles currently resident on the news server, and an estimate of the length of time needed to compute the checksum on a Sun 3/50. On average, computing the integrity checksum takes 3 or 4 seconds, except where sources are binaries are involved; as those files are larger, the computation takes around 20 to

³The entries *binaries* and *sources* are for all binary and source newsgroups in *comp*, and the group *gnu.emacs.sources*.

newsgroup	min size		max size		avg size	
	bytes	time	bytes	time	bytes	time
binaries	489	0.75	96393	148.25	17116	26.32
comp.bugs.4bsd.ucb-fixes	698	1.07	2875	4.42	1504	2.31
control	297	0.46	73746	113.42	1786	2.75
sources	573	0.88	55338	85.10	12454	19.15
bionet	393	0.60	91340	140.48	3289	5.06
comp	358	0.55	180402	277.45	2470	3.80
dartmouth	308	0.47	5538	8.51	683	1.05
gnu	324	0.50	81374	125.15	2422	3.73
misc	235	0.36	51512	79.22	1975	3.03
sci	320	0.49	102682	157.92	2220	3.42

Figure 5: Estimated Time (in Seconds) to Compute Integrity Checksum

25 seconds. (Encrypting the integrity checksum and data encryption key using the RSA cryptosystem would add approximately 1 to 3 seconds, assuming a good implementation of RSA [LAUR90].) Given that the four entries in the top of the table represent groups in which forged postings, or altered postings, represent a substantial threat, the penalty seems well worth the safety gained. With other newsgroups, safety is not a paramount issue, and so the need for using mechanisms to ensure integrity and authenticity is less immediate; nevertheless, considering that most news posting is done in batches, there is usually an appreciable delay from the posting to the transmission to other sites. Adding a few seconds more hardly seems like an excessive burden.

Comparing and Contrasting Other Approaches

Currently, the news server *nntp* offers a simple "authorization scheme" for posters. Such a scheme authenticates the poster, checks to see if the poster is authorized to post, and if so accepts the article. The authentication is only to the *nntp* server, and in no way is that authentication bound to the article posted. Nor can this scheme be easily altered to provide that binding for two reasons. First, there is no provision for integrity checking, so articles can be altered in transit without detection. Second, as the mechanism involves putting a password known to *nntp* and the authorized poster, news readers would have to trust the *nntp* server and all intermediate nodes. This is quite unreasonable in a network as diverse as USENET.

An alternate approach would be to use Kerberos [STEI88] in combination with an integrity-checking algorithm to authenticate the poster. The Kerberos scheme requires that all users be registered on a central server, and herein lies the problem: all parties must trust that server to be physically secure and not penetrable by attackers; of course all administrators and users with access to it must be trusted as well.⁴ Within a single organization, such trust is possible; on the USENET, which is a decentralized network of autonomous entities, it is highly unlikely that any organization would agree to trust an authentication server which it does not physically control. To trust a server under either's control, or to trust any single entity (or set of entities) on the network. For this reason, Kerberos is a viable alternative for individual organizations, but much less so for many different organizations.

Using the international standard X.411 [X41187] would require modifying the underlying transport agents (such as *nntp* or *rnews*) to know about the security-related headers. This conflicts with the goal of making the modifications to existing

news software be as unobtrusive as possible, and (in particular) have them only at the news reading and/or posting level. Hence the X.411 scheme hence does not meet this goal.

Conclusion

This paper has shown that adding authentication and integrity mechanisms to the network news is possible, and using a mechanism similar to those used for privacy-enhanced electronic mail enables the USENET to piggyback onto an infrastructure that is expected to become part of the Internet soon. Further, those mechanisms do not impact the performance of news too severely.

The main disadvantage of this scheme is that the public-key cryptosystem which will be used for electronic mail, the RSA cryptosystem is covered by patents administered by RSA Data Security, Inc., and so many people will have to pay a fee of some kind for certificates⁵. Hence if the same cryptosystem were to be used for news, licensing issues would arise. We should note that, from a technical point of view, the same certificates will function equally well for both news and electronic mail. An alternative is to use some other public-key cryptosystem; however, the RSA system has distinct advantages, among them being recommended for use by international standards, being supported by an infrastructure that will soon be available, and being completely compatible with electronic mail. It is also (relatively) efficient to implement, and is believed to be very strong cryptographically.

In the end, the users and administrators will decide whether the authenticated, integrity-checked news articles are worthwhile. The mechanism described in this paper is quite flexible and robust enough to meet the needs of those who want authenticated, integrity-checked news of those who only want some of the news articles to be authenticated. Those who do not want such a mechanism can still read all articles, authenticated or not; and those who do want the mechanism can also read all articles (but believe only the authenticated ones, we hope).

Acknowledgements

My thanks to the Privacy and Security Research Group, who developed the privacy-enhanced mail proposals, and to Holly Bishop, who found Chernenko's exact title, and to Robert Van Cleef and Gene Spafford, who helped reconstruct the

⁴Contrast this to the *nntp* mechanism, which also requires that all intermediate nodes and links be trusted.

⁵If RSA Data Security, Inc., issues the certificate, the certificate will be good for two years and will cost \$25 (of which \$22.50 will be a service charge). An organization may issue its own certificates, good for two years, using special equipment, and then would pay \$2.50 per certificate. See [BISH90] for more details. Note that the patent does not require the U.S. government to pay royalties.

USENET logic bomb incident.

References

- [BEER84] P. Beertema, "USSR on Usenet: The Mystery Unravels," message <5779@mcvax.UUCP> (Apr. 15, 1984).
- [BISH88] M. Bishop, "An Application of a Fast Data Encryption Standard Implementation," *Computing Systems* 1(3) (Summer 1988) pp. 221-254.
- [BISH90] M. Bishop, "Privacy-Enhanced Electronic Mail," Technical Report PCS-TR90-150, Revision 1, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH 03755 (June 1990).
- [DENN82] D. Denning, *Cryptography and Data Security*, Addison-Wesley Publishing Company, Reading, MA (1982).
- [FIPS80] Federal Information Processing Standards Publication 81, *DES Modes of Operation* (Dec. 1980).
- [HORT87] M. Horton and R. Adams, *Standard Format for Interchange of USENET Messages*, RFC-1036 (Dec. 1987).
- [KENT89] S. Kent and J. Linn, *Privacy Enhancement for Internet Electronic Mail: Part II - Certificate-Based Key Management*, RFC-1114 (Aug. 1989).
- [LAUR90] D. Laurichesse, *Mise En uvre Optimisee du Chiffre RSA*, Technical Report LAAS-90052, Laboratoire D'Automatique et D'Analyse des Systemes (Mar. 1990).
- [LINN89a] J. Linn, *Privacy Enhancement for Internet Electronic Mail: Part I - Message Encipherment and Authentication Procedures*, RFC-1113 (Aug. 1989).
- [LINN89b] J. Linn, *Privacy Enhancement for Internet Electronic Mail: Part III - Algorithms, Modes, and Identifiers*, RFC-1115 (Aug. 1989).
- [POST82] J. Postel, *Simple Mail Transfer Protocol*, RFC-821 (Aug. 1982).
- [RIVE78] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM* 21(2) (Feb. 1978) pp. 120-126.
- [STEI88] J. Steiner, C. Neuman, and J. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *USENIX Conference Proceedings* (Winter 1988) pp. 191-202.
- [THOM74] R. Thomas, *On the Problem of Signature Authentication for Network Mail*, RFC-644 (July 1974).
- [VOYD83] V. Voydock and S. Kent, "Security Mechanisms in High-Level Network Protocols," *Computing Surveys* (June 1983) pp. 135-171.
- [X41187] CCITT Recommendation X.411, *Message*

Handling Systems: Message Transfer System: Abstract Service Definitions and Procedures (1987).

- [X50087] CCITT Recommendation X.500, *The Directory - Overview of Concepts, Models, and Services* (1987).

Matt Bishop is an assistant professor in the Department of Mathematics and Computer Science at Dartmouth College. His interests include computer security, cryptography, operating systems, user interfaces, and classical history. He is a member of the Privacy and Security Research Group and has chaired the last two UNIX Security Workshops. He may be reached at Matt.Bishop@dartmouth.edu, or by U.S. Mail at Department of Mathematics and Computer Science, Bradley Hall, Dartmouth College, Hanover, NH 03755 USA.



An Experimental Implementation of Draft POSIX Asynchronous I/O

A. Lester Buck†, Robert A. Coyne, Jr. – IBM Federal Sector Division, Houston

ABSTRACT

As UNIX moves into the large systems environment, the need for an efficient asynchronous input/output (I/O) mechanism becomes more acute. We review various designers' approaches, absent existing standards, to graft either non-blocking or fully asynchronous I/O capabilities onto UNIX. We then describe an experimental implementation of asynchronous event notification and asynchronous I/O for AIX/370 based on unapproved Draft 9 of the POSIX 1003.4 standard. Our goal is a high performance I/O system that can efficiently handle hundreds of outstanding I/O operations in a single server process, with dozens of operations transferring data simultaneously. We describe queueing structures, changes to standard kernel routines, and extensions to the device driver interface to integrate asynchronous functionality. Our first implementation supports only raw transfers directly between devices and user buffers. Issues of real memory consumption, page faults in interrupt context, write ordering semantics, security, asynchronous *ioctl*(s), and compatibility with *select*() are discussed.

Introduction

The UNIX operating system was originally implemented on minicomputer hardware. As it has proven itself in the marketplace, it has been extended to both smaller and larger systems. Large computing installations are characterized by very heavy input/output loads with significant investment in high performance I/O devices and subsystems. As UNIX moves into this large systems environment, the need for an effective, efficient means of overlapping multiple I/O operations and computing for a single application process becomes more acute. For example, the largest IBM System/390 mainframe may have up to 256 fiber optic I/O channels, each capable of transferring at 10 MB/sec, and might support an aggregate sustained I/O bandwidth over 500 MB/sec from the bare hardware [46]. Our goal was to implement a high performance I/O facility in AIX/370 capable of efficiently managing hundreds of outstanding operations with dozens of operations transferring data simultaneously. We describe here an experimental implementation of such a facility based upon unapproved Draft 9 of the proposed POSIX P1003.4 realtime extensions standard [18].

The structure of this paper is as follows: The next section covers some background on the common meanings of asynchronous and non-blocking I/O. We then present a short history of overlapped computation and I/O from both the hardware and software perspectives. Following that section is a survey of implementations of non-blocking I/O in various versions of UNIX. Two sections summarize the application programming interfaces (API) of the

Draft 9 standard as it relates to asynchronous event notification and asynchronous I/O. Subsequent sections discuss the design issues of our implementation and give details about our implementation of asynchronous event notification.

To place asynchronous I/O in context, we trace the path taken by a typical synchronous I/O operation as it travels through UNIX and follow the path of an asynchronous I/O operation in our implementation. More details of our asynchronous I/O implementation are in sections that follow, including asynchronous *ioctl*(s). Finally we discuss some outstanding issues found in integrating asynchronous I/O into UNIX, and we summarize the current status of this work and state our conclusions.

Terminology

The word *asynchronous* simply means "not synchronous." The term *synchronous* is defined as [30]: (1) happening, existing, or arising at precisely the same time, (2) recurring or operating at exactly the same periods, or (3) having the same period and phase. In computing, the word *synchronous* refers to occurrences¹ that have a fixed relationship at certain points in their execution sequence. At well-defined synchronization points, the time relationship between synchronized occurrences is fixed before the processes associated with these occurrences are allowed to proceed. If necessary, one or more of the associated processes may be blocked (i.e., suspended from its execution) until the synchronization condition is met.

†UTSI, Inc.

¹We avoid the word *event* here for obvious reasons.

In real systems, such as UNIX, there are many synchronization conditions, and we must be clear as to what type of synchrony we refer. In one sense, all normal UNIX I/O to all types of files is basically synchronous, in that the application process blocks until the system is finished transferring data to or from the application buffer. The synchronizing condition is the completion of the read/write system call. But a *write()* system call in normal UNIX that uses the system buffer cache has a second asynchronous aspect, in that the physical write to the device media may be delayed for some unknown period [37]. The *FSYNC* file mode and the *fsync()* system call are used to give strong hints to the system about when to schedule the physical transfer from the buffer cache to the device. This second type of synchronous I/O scheduling is discussed briefly in a later section, but we concentrate in this paper on the first sense of synchronous or blocking I/O.

Synchronous behavior occurs at many levels in computing. At the lowest hardware level, synchronous data transfer may be over a communication link, an I/O bus, or a central processor backplane, where the synchronization condition is imposed by a hardware clock. At the higher software levels, there is an extensive literature on the subject of co-operating sequential processes [4, 9].

An asynchronous relationship between two occurrences means there is no direct time relationship. Occurrence A might happen before occurrence B, or after, with no direct effect on the forward progress of the processes associated with occurrences A and B. The world is filled with asynchronous relationships, and these are reflected in common computing concepts. One pure example of an asynchronous occurrence is a hardware interrupt that might ultimately be generated by a transition on a control line. By its very nature, an interrupt is decoupled in time from other occurrences in the computing system. Higher level software layers may present asynchronous occurrences to an application, the most common being a UNIX signal. The theoretical basis of asynchronous interactions between processes is more limited than that for synchronous interactions. Wettstein and Merbeth [45] have attempted a systematic development of the concept of asynchronization between parallel processes.

Asynchronous and non-blocking I/O are analogous to interrupt driven and polled device drivers, respectively. In the typical situation, where interrupts are much more efficient than polling, a non-blocking I/O requires a system call, with attendant overhead, to check on the status of a request. More efficient is a non-blocking I/O, where the application may poll a user space completion variable. Where interrupts are supported, the interrupt handler may be required to poll in the interrupt routine to find the interrupting device. In the most efficient case, the

interrupt is tagged with a unique identifier, eliminating all polling. True asynchronous I/O does not require any form of polling to handle I/O completion.

Clearly the Unix kernel overlaps I/O and computation on a systemwide basis, even though a given process is blocked waiting for I/O completion. If a process uses the buffer cache, the kernel is able to perform write-behind of output buffers and, when it detects a sequential access pattern, it also performs block read-ahead. In this way, a write request may often return immediately after the data block is copied into a kernel buffer. A sequential read may return immediately if the input device has had sufficient time to respond to the kernel's read-ahead request. This behavior is not guaranteed, being critically dependent on availability of free system buffers and on system load. For high performance I/O, the extra copy required through the kernel buffer and the restriction of I/O transfer size to units of a kernel buffer size make this facility unattractive. For real-time response, the nondeterministic process blocking is completely unacceptable.

Simulated Asynchronous I/O

While common versions of UNIX do not support asynchronous I/O, this does not mean that applications cannot overlap computation and I/O operations. A single process performing synchronous I/O blocks until completion, but an application may be composed of a collection of processes. A common method for managing overlapped I/O is to funnel all I/O requests through a master process, that in turn communicates with a set of drone processes through shared memory [37]. Each drone exists to start one synchronous I/O request for the master process, block until complete, and report completion status back to the master. This is an effective method of achieving the benefits of asynchronous I/O in standard UNIX and is commonly used by I/O intensive applications. Unfortunately, this scheme does not scale well in the number of I/O processes, as it requires a process context switch for each I/O operation. On a large system with a large number of I/O drones, a point is reached where the processor is overloaded from switching uselessly between blocking processes while the I/O subsystem has idle capacity.

Another approach to simulate asynchronous I/O with standard UNIX employs a dedicated disk handling process to communicate with a server's lightweight processes by means of shared memory [33]. The disk process manages its own queue of requests and discharges them in order, blocking while they are active. The server's lightweight processes enqueue and dequeue disk requests without waiting for the current request to terminate. The limitation of this scheme is the inability of the user disk process to have the next request already waiting for

initiation on the disk driver queue when the current request completes.

Overlapped Computation and I/O

The earliest computers had very primitive I/O facilities that required continuous and detailed control by the central processor. The CPU was effectively the controller for all devices. As computers evolved, it became possible to begin to overlap processing and I/O in a primitive way by interleaving I/O instructions with general calculations. For example, the IBM 701 electronic data processing machine[40] could perform general calculations between I/O transfers so long as it had issued a blocking copy-and-skip instruction just before the device was ready to transfer each word of data. This was clearly a timing-critical facility that demanded careful programming and was not feasible for use in an operating system.

Knuth[21] credits the original implementation of direct memory access (DMA) with overlapping computation, including interrupts, to the DYSEAC computer[24] in 1954, while Rosen[38] attributes the innovation of independent data channels to the IBM 709, delivered in 1958. The 709 was able to timeshare the core memory between the central processor and as many as six data channels. van de Goor[43] presents an excellent summary of the evolution of I/O management, from early CPU control through today's intelligent controllers that overlap computation and I/O internally.

Once computer systems were freed from performing the low level functions of a device controller, various innovations in operating systems could be introduced. Buffering, which is the technique of overlapping computation time and I/O time, is a crucial facility offered by operating systems to shield programs from the speed mismatch between the central processor and typical I/O devices. In the simplest case, two buffers may be switched off so that one is being emptied while the other is being filled. In realistic computing environments, considerably more than two buffers may be effectively utilized to smooth out variations in computing time or I/O time per block, or to hide environmental effects when the I/O channel is being shared among several processes and is not always immediately available. Multiprogramming operating systems evolved in part to keep the I/O subsystem performing useful work when the I/O needs of an individual program were satisfied. I/O devices could be multiplexed among several programs at once, servicing each program's buffers as necessary and avoiding unnecessary idle time. Knuth covers the basic buffering algorithms[21] and some of the practical considerations for external sorting to tape and disk[22]. Madnick[25] presents extensive details of the I/O subsystems on System/370, including example I/O channel programs, interrupt processing, and buffering

routines. Freeman[11] gives an excellent explanation of I/O access methods and the levels of service offered in traditional large mainframe operating systems, including buffering and asynchronous I/O.

Previous Implementations

Many operating systems include support for asynchronous or non-blocking I/O in some form, from MVS[15], through VMS[7] and RSTS[28], to realtime executives of all types. In this section, we discuss a few existing implementations of asynchronous or non-blocking I/O in versions of UNIX. We do not intend this as an exhaustive survey, a detailed review, or a critique of other work. Instead we present this information as a set of interesting ideas and evidence of the diversity of approaches taken by various designers in solving a common problem, absent existing standards.

Various facilities exist under commonly available versions of UNIX to allow an application to do non-blocking I/O[41]. Stock System V includes the *poll()* system call and various terminal driver settings. Berkeley 4.3BSD supports *select()* and SIGIO. The *select()* and *poll()* functions allow the application to poll for available input on a file descriptor. System V limits *poll()* to STREAMS devices, such as network connections and, more recently, terminals. BSD *select()* supports polling on sockets and terminals. The SIGIO mechanism would be a very useful mechanism for asynchronous I/O, except it cannot tag outstanding requests and signals are not queued in 4.3BSD. Neither of the two main branches of UNIX support these facilities for disk or tape. In addition, the semantics of *select()* are weak for overlapping I/O, as the application may still be suspended for a large block of output or while reading more than the available input buffer.

Pyramid Technology Corporation, in their OSx version of UNIX, has implemented an *ioctl()* interface to a raw² disk driver that supports non-blocking I/O[36]. An application allocates a fixed set of buffers that are page-locked by a DKIOCMLOCK *ioctl*. It then issues an *ioctl*(fd, DKIOCASTRT, *aiobuf*), where *aiobuf* points to a structure that includes a command (read, write, ordered write), a disk address, a user buffer address, the number of bytes to transfer, and an (application defined) user context reference pointer. The application may poll for completion by issuing an *ioctl*(fd, DKIOCSTAT, *aiostatbuf*), where *aiostatbuf* is an array of status structures describing completed requests. The returned status includes the completion status, buffer size, user buffer address, and user context reference

²The term *raw* is used when a block device, such as disk, is accessed through the character special read/write interface, rather than the usual *ddstrategy()* interface.

pointer. Also provided is an *fcntl()* FRAIOSIG facility that will issue a SIGEMT signal upon I/O completion. Since SIGEMT cannot specify the completed request, if multiple requests are outstanding a DKIOCSTAT ioctl is still required. The minimal changes to process and memory management semantics that guarantee system integrity are specified: a shrinking *brk()* and *shmdt()* return EBUSY for memory locked by DKIOCMLOCK, *exec()* returns EINVAL if there are any outstanding requests, and *exit()* waits for asynchronous I/O completion and unlocks all memory locked by DKIOCMLOCK. AT&T has delivered an implementation of this interface in a version of System V Release 3.

Convex Computer Corporation, in their ConvexOS, designed a scheme that allows setting a flag for a file descriptor that converts all subsequent *read()/write()* requests into asynchronous operations. The new *asiostat()* system call returns the number of bytes transferred on the given descriptor since the last *asiostat()*, or -1 if an error has occurred. No request handles are returned and no status blocks are available to the application. Their implementation uses *asiodaemon* processes to block for each asynchronous request. Harris Computer Systems CX/UX operating system offers an asynchronous I/O capability through *aread()* and *awrite()* calls [14], that return a 32-bit I/O identifier for each operation. Harris supports various forms of completion notification: waiting, polling, or receiving a signal. An application is restricted to 32 outstanding asynchronous requests per file descriptor.

UNICOS, the version of UNIX from Cray Research, Inc., for their supercomputers, offers a full version of asynchronous I/O, including support for regular files [5]. The *reada()* and *writeta()* calls accept a file descriptor, a user buffer address, a byte count, a completion status block, and a signal number as arguments. The completion status block includes a one-bit flag marking a completed request, an error number, and a count of total bytes transferred. Several requests may share a given signal number, and the signal handler must scan for all possible completions. UNICOS assists in this function by restarting the handler if new completions have arrived on exit. A *listio()* function is included for initiating a list of requests. The *listio()* call may either return immediately or wait for all requests to complete. An especially useful feature of the UNICOS *listio()* is the ability to specify a stride into a disk file and user memory, thereby allowing a high performance scatter/gather operation matched to the vector processing system, critical to out-of-core calculations [34].

SunOS Release 4.1, from Sun Microsystems, Inc., offers a library interface to internal asynchronous functions through *aioread()*, *aiowrite()*, *aiowait()*, and *aiocancel()* [42]. The underlying mechanism is not publicly disclosed. By publishing

only a library interface, Sun is free to manipulate the base asynchronous system call interface mechanism as necessary for eventual POSIX compliance, without supporting obsolete system calls indefinitely. The arguments to *aioread()* and *aiowrite()* take a file descriptor, a user buffer pointer, a transfer byte count, an offset and whence, and a pointer to a user result structure. Completion notification may be taken synchronously using *aiowait()* or asynchronously by receiving a SIGIO signal. The signals are delivered reliably and are queued as necessary, so notifications are not lost. SIGIO is somewhat overloaded, as several requests may be queued for one SIGIO and a file descriptor may generate SIGIO independently of an asynchronous request. Sun's database accelerator package improves the performance of their standard asynchronous I/O routines. The implementation handles both regular and special files.

Modular Computer Systems, Inc. (ModComp) has taken the base code of System V Release 3 and completely reworked it into a fully preemptive kernel that is still System V Interface Definition compliant, while adding numerous features for realtime support including events and asynchronous I/O [31] [12]. The result is the REAL/IX operating system which is available for their 68030-based hardware platforms. ModComp has closely tracked the work of P1003.4 and their events and asynchronous I/O are quite similar to the draft. Asynchronous I/O is supported to both extent-based (i.e., contiguous) regular files and character special files. I/O to extent-based files bypasses the kernel buffer cache. An *fcntl()* flag may be set that allows asynchronous requests to be emulated by a synchronous I/O operation when required, which decouples application and driver development and removes data size and alignment rules for applications that cannot comply. The *aread()* and *awrite()* system calls accept an extra *aioch* structure that describes the user's buffers, number of bytes to transfer, offset and whence, and other parameters. A new *aio()* entry point was added to *cdevsw* that receives a kernel data structure, *areq*, describing the operation and arguments. Neither the application nor the driver may block at any time during asynchronous I/O in REAL/IX, so asynchronous I/O user buffers must be locked in physical memory before a request is initiated. Regular files are supported through the SVR3 File System Switch. Overhead for locking and unlocking buffers for each asynchronous operation are undesirable in a realtime system due to possible page faults, so REAL/IX supports a per-process cache of locked pages on the presumption that a process will reuse its asynchronous I/O buffers.

Concurrent Computer Corp. sells a realtime version of UNIX known as RTU. RTU has implemented an asynchronous event notification facility based on asynchronous system traps (ASTs) [27],

which are an enhancement to the UNIX signal mechanism and are one of the existing implementations upon which the P1003.4 committee based its events design. RTU supports a large number of distinct user-defined ASTs, with an attached user-specified parameter. ASTs are designed to simulate hardware interrupts, with reliable, queued handling by priority level. Various forms of asynchronous I/O may be implemented by the drivers using the AST mechanism. A typical driver accepts an asynchronous request through a private *ioctl()* call that passes a driver specific request structure [10]. The driver is responsible for locking and unlocking physical memory and posting the AST on completion. Only raw devices are supported for asynchronous I/O.

Digital Equipment Corp. in its ULTRIX product has implemented an elegantly simple form of non-blocking I/O called *nbuf* [8]. A process announces to the system that it will be performing raw I/O using *N* buffers. Once *N-buffered* operation is enabled, *read()* and *write()* are not guaranteed to

have completed when they return to the user process, although the return value is the number of bytes requested and the file pointers are updated. The *nbuf* scheme depends on a set of *ioctl()* operations and on *select()* support in the disk and tape drivers. An *ioctl(fd, FIONBUF, count)* enables *count* buffers to be used with the raw device associated with the file descriptor. If *count* is zero, *N-buffering* is terminated and incomplete requests are waited on. At some time after initiation, an *ioctl(fd, FIONBDONE, buffer)* is performed to retrieve the actual return value for the given buffer's request. *FIONBDONE* either blocks until completion or, if *FNDELAY* has been specified, returns *EWouldBlock* if the request is not yet complete. The status of the request may be polled or waited on by using *select()*. The *select()* call returns immediately if less than *count* operations have been started on an *nbuf* channel, or blocks waiting for a request to complete.

Portion of new header file <events.h>:

```
#define _NEVT ((EVTCLASS_MAX - EVTCLASS_MIN) + 1)
#define _EVTSETSZ ((_NEVT+31)/32) /* # of longs in event classmask array */

typedef long evt_class_t;

/* event classmasks are specified thus */
typedef struct {
    int setsize; /* basically a version number */
    unsigned long evts[_EVTSETSZ]; /* mask words, 32 event classes each */
} evtset_t;

struct event {
    void (*evt_handler)(); /* event notification function */
    void *evt_value; /* application dependent value */
    evt_class_t evt_class; /* event class for this event */
    evtset_t evt_classmask; /* classes blocked during handler */
};
```

New elements in struct proc, <sys/proc.h>:

```
evtmask_t p_evtmask; /* current event mask */
struct kevent *p_evthead[_NEVT]; /* head of pending event list */
struct kevent *p_evttail[_NEVT]; /* tail of pending event list */
evtmask_t p_evt; /* event classes pending */
evt_class_t p_sigclass[NSIG]; /* signal event class */
evtmask_t p_evtpoll; /* classes being polled */
int p_evtpending; /* total pending events */
```

New elements in struct user, <sys/user.h>:

```
evtmask_t u_evtonstack; /* event classes to take on sigstack */
evtmask_t u_evtintr; /* event classes that intr syscalls */
evtmask_t u_oldevtmask; /* saved mask from before evtpoll */
struct timeval u_evtpollto; /* time out limit for event poll */
```

Figure 1: Asynchronous event notification header additions

Even sophisticated end users may implement asynchronous I/O if they have kernel source available [13]. A group at Lawrence Livermore National Laboratory heavily modified the Amdahl UTS kernel to include a lightweight kernel tasking facility based on *setjmp()/longjmp()*. They then implemented non-blocking kernel process I/O by having their tasking library queue a process' requests until all tasks are either waiting for I/O or are quiescent. All the process' requests are then submitted using a pseudo device driver call, which only then blocks the process. On completion of any of the requested I/Os, the kernel process is unblocked and allowed to run again.

Asynchronous Event Notification API

This section and the following summarize the important aspects of Draft 9 asynchronous event notification and asynchronous I/O, with some minor commentary that should help place the new functions in context.

event Definition Structure

The asynchronous event notification API is designed to enhance the traditional UNIX signal mechanism because: (1) events are reliable, while POSIX.1 signals may be lost, (2) events are tagged and therefore may carry data, (3) all events are user definable, expanding on the limited SIGUSER1 and SIGUSER2, and (4) the order of event delivery is

specified, while signal delivery order is not standardized³ [17]. Events were designed to represent occurrences that are the result of some activity in the system, including: (1) asynchronous I/O completion, (2) timer expiration, (3) message arrival, or (4) user defined event occurrence [i.e., *evtraise()*]. The goal of the events API was notification in a uniform and reliable manner. Subsequent to Draft 9, the P1003.4 committee has moved toward using an enhanced signal interface to achieve most of these goals. Since asynchronous I/O is not particularly sensitive to the notification method used, and our events implementation was essentially complete when this shift occurred, we followed Draft 9.

Events are built around an event definition, as specified in an *event* structure as shown in Figure 1. The *evt_handler()* function specifies the event notification function, completely analogous to the signal handler function of POSIX.1. The *evt_value* element is an arbitrary piece of data that tags the given event and may point to arbitrary user data. The application uses the *evt_value* parameter to track which event is being handled. The *evt_class* specifies the event class for the event and would typically be a small integer. Events are delivered in strict class order, so the class is a type of priority. The *evt_classmask* determines which event classes

³AIX/370, for example, delivers SIGFPE first, then the rest of the signals in ascending numerical order.

```
/* manipulate event class sets */
int      evtemptyset(evtset_t *set);
int      evtfillset(evtset_t *set);
int      evtaddset(evtset_t *set, evt_class_t class);
int      evtdelset(evtset_t *set, evt_class_t class);
int      evtismember(evtset_t *set, evt_class_t class);

/* set or get the mask of currently blocked event classes */
int      evtprocmask(int how, evtset_t *set, evtset_t *oset);

/* suspend process, blocking specified event classes */
int      evtsuspend(evtset_t *evtmask, struct timespec *timeout);

/* poll for event notification, taking unblocked events asynchronously */
int      evtpoll(evtset_t *evtmask, struct timespec *timeout,
                void **value, evt_class_t *class);

/* generate an application defined event for the calling process */
int      evtraise(struct event *eventp);

/* associate one or more signals to an event class */
int      evtsigclass(sigset_t *mask, evt_class_t class);

/* non-local jumps */
int      evtsetjmp(evtjmp_buf_t *env);
int      evtlongjmp(evtjmp_buf_t *env, int val);
```

Figure 2: Asynchronous event notification functions

are blocked from delivery when (and if) the event notification function executes.

Event delivery has the required deterministic behavior. Within a class events are delivered in FIFO order, and across unblocked classes they are delivered in strict order of descending class. Once

Additional elements in the proc structure, <sys/proc.h>:

```
int p_aioqueued;           /* total of all aio's on any queue */
int p_aioactive;           /* total of aio's active on any file */
int p_aiodone;             /* total of aio's awaiting errno post */
struct kaiocb *p_aio;      /* list of aio's awaiting errno post */
int p_aiopagelock;         /* total number of pages with aio locks */
int p_liocount;            /* # of LIO_WAITing aio's active */
```

Portion of new header file <aio.h>

/* kernel asynchronous I/O queue element */

```
struct kaiocb {
    char      *aio_buf;           /* user buffer address */
    unsigned   aio_nbytes;        /* requested number of bytes to transfer */
    unsigned   aio_nobytes;       /* number of bytes actually transferred */
    int        aio_whence;        /* how to generate desired offset */
    off_t      aio_offset;        /* offset argument */
    int        aio_errno;         /* EINPROG while active, errno on completion */
    int        aio_prio;          /* asynchronous I/O operation priority */
    struct event aio_event;       /* event to be posted upon I/O completion */
    int        aio_flag;          /* post event, etc. */
    int        aio_ext;           /* AIX extended argument (readx, writex) */
    struct kaiocb *aio_kernaddr; /* kernel address of queue element */
    /*** user level struct aiocb ends here ***/
    struct kaiocb *aio_before;    /* next older aio */
    struct kaiocb *aio_after;    /* next younger aio */
    off_t      aio_mpxchan;       /* AIX multiplex channel */
    long       aio_fmode;         /* file mode bits */
    struct file *aio_fp;          /* file block pointer */
    struct proc *aio_proc;        /* process handle that issued aio */
    struct aiocb *aio_useraddr;   /* user virtual address of this aiocb */
    void       (*aio_strat)();    /* strategy routine that queues this aio */
    int        (*aio_mincnt)();   /* routine that limits physical io */
    struct buf *aio_bufhead;      /* buffer header */
    int        aio_npf;           /* page frames locked for this operation */
    lioasync_t *aio_lioasync;     /* associated LIO_ASYNC count and event */
    int        *aio_suspend;     /* pointer to iosuspend done flag */
    int        aio_suspend;       /* index into iosuspend'ed aiocbp array */
};

/*
 * For asynchronous ioctl operations, the following
 * fields are redefined to supply the ioctl arguments.
 */
#define aio_cmd      aio_whence    /* ioctl command */
#define aio_argp     aio_buf       /* argument pointer */
#define aio_opcount  aio_offset    /* operation repeat count */
#define aio_kargp    aio_bufhead.b_un.b_addr /* kernel copy of arg data */
#define aio_argsize  aio_nbytes    /* sizeof(arg data), stored by kernel */
```

Figure 3: Asynchronous I/O header additions

queued, they must be delivered reliably, so the system must reserve whatever resources are necessary when an application either explicitly [*evtraise()*] or implicitly [*aread()*, *awrite()*, etc.] solicits an event for subsequent delivery.

The API becomes more complicated when specifying the semantic relationship between signals and events. All signals default to an implementation chosen event class, *EVTCLASS_SIG*, that is not maskable by the event classmask. Signals in *EVTCLASS_SIG* always interrupt event notification functions. A process may reassign any signal, except *SIGKILL* and *SIGSTOP*, to any class, and thereby determine the delivery order of normal signals. Multiple signals pending within an event class are delivered according to POSIX.1, and the order of delivery of mixtures of unblocked signals and events in a single class is undefined.

The process semantics of events are straightforward. The current event mask is copied to the child on *fork()* and set to block all events on *exec()*. The queue of pending events is discarded on both *fork()* and *exec()*. The mapping of signals to event classes is copied on *fork()* and reset to the default on *exec()*. All blocked pending events are discarded on *exit()*. Event notification functions are somewhat different from signal handlers, as they are only activated by the application explicitly soliciting an event, while signal handlers respond to myriad system occurrences with no application initiation. While the event handler is executing, the previous event classmask is saved and a new mask is formed from the union of the current classmask, the mask in the event definition and the class of the event notification. When the event handler returns, the saved classmask is restored.

Asynchronous Event Functions

Many of the asynchronous event notification functions specified in the P1003.4 Draft 9 are analogs of the POSIX.1 [17] signal functions [Figure 2]. For manipulating events, the draft defines the basic *evtemptyset()*, *evtfillset()*, *evtaddset()*, *evtdelset()*, and *evtismember()* functions that are analogous to the signal mask manipulating functions. The *evtprocmask()* function is the equivalent of the standard *sigprocmask()*. The *evtsuspend()* function adds new flexibility to the analog of *sigsuspend()* for events by including a timeout argument, with semantics analogous to *select()* (i.e., poll, indefinite wait, or timeout).

An entirely new function is *evtpoll()*, that allows the application to take unblocked events asynchronously (via the event notification function), while also suspended waiting to synchronously process an event from another set of blocked, polled event classes. The new behavior is for an unblocked event handler to run to completion, but the

application again suspends waiting for one of the events being polled, or until the specified timeout elapses. Upon successful return, *evtpoll()* returns the *evt_value* and *evt_class* elements of the event definition structure, but ignores the *evt_handler* and *evt_classmask* elements.

The analog to POSIX.1 *kill()* is the *evtraise()* function for posting an event to the application. Note that *evtraise()* in particular, and events in general, have no provision for delivering events between processes. The event machinery is purely for intraprocess use and is not meant to be another mechanism for interprocess communication.

The *evtsetjmp()* and *evtlongjmp()* functions are extensions of *sigsetjmp()* and *siglongjmp()* that include the event classmask in the stored state of the process. The *evtsigclass()* function associates a set of signals contained in the mask parameter to a given event class.

Asynchronous I/O API

*aio*cb Control Block

The goal of asynchronous I/O is to allow an application to explicitly overlap computation time and I/O time. A single process may then simultaneously perform I/O to a single file multiple times or to multiple files multiple times. The asynchronous I/O API is built around an asynchronous I/O control block, *aio*cb structure (see Figure 3). Note that, although the functionality is almost identical with the draft, this layout of the *aio*cb and associated function interface is slightly different from that contained in the draft standard. We discuss these minor differences below with our implementation. The *aio_buf* member is a pointer to the user buffer where data will be transferred to [from] by *aread()* [*awrite()*]. The *aio_nbytes* member is the requested number of bytes to transfer. The *aio_whence* and *aio_offset* parameters are analogous to the *whence* and *offset* arguments defined by *lseek()*. The implied *lseek()* is performed before every successful return of an asynchronous I/O function (i.e., a successfully queued request), including incrementing the file offset by the number of bytes that will be transferred if the request is eventually successful.

The *aio_errno* parameter is the simplest and most direct completion flag. It is set to *EINPROG* if the request is successfully queued by the system and, on completion of the request, it is set to zero or to an appropriate error code. The *aio_nbytes* is the number of bytes actually transferred by the completed request and is an output from the system. If *aio_errno* reflects an error condition, then *aio_nbytes* is set to -1. The *aio_prio* member carries the asynchronous I/O operation priority, a hint to the system on how to order asynchronous requests. The *aio_prio* member has no effect on process scheduling priority, as it applies only to the I/O

subsystem. The meaning of *aio_prio* is implementation-defined, and the draft specifies that the implementation must document the order of the asynchronous I/O operations. The *aio_event* structure is a definition of the event that will be posted on completion of the request if the *AIO_EVENT* bit is set in the *aio_flag* member. Otherwise, no event will be posted on completion.

The process semantics of asynchronous I/O are reasonably straightforward. Asynchronous I/O requests are not inherited across *fork()*. On *close()*, for a given file descriptor, and on *_exit()* and *exec()* for all of a process' open file descriptors, an attempt is made to cancel and dequeue active asynchronous requests queued by the current process. Non-cancelable operations are waited on and associated event notifications are suppressed. Asynchronous I/O requests must be guaranteed never to affect memory outside the requesting process, even if the process exits or closes the file descriptor before the I/O is complete or detaches a shared memory buffer. The behavior of all asynchronous I/O functions that transfer data is undefined if the buffer pointed to by *aio_buf* becomes an invalid address during the time the operation is active or if multiple outstanding operations are using the same *aiocb* structure. The definition of cancelable operation may vary from device to device.

Asynchronous I/O Functions

The *aread()* function accepts a file descriptor and a pointer to an *aiocb* and queues the request for asynchronous execution [Figure 4]. The function returns immediately when the read request has been initiated or queued for later execution. If an error condition prevents the request from being queued, then the function call returns without having initiated or queued the request. Once a request has been successfully queued, the function returns and further errors, if any, are reflected in the *aio_errno* and *aio_nbytes* parameters, with event notification if requested. The *awrite()* function is identical, except

for the direction of data transfer and the requirement to ignore *aio_whence* and *aio_offset* when the descriptor has the *O_APPEND* file status.

The *listio()* function allows a process to initiate a list of asynchronous I/O operations with one system call. This facility is similar to the BSD *readv()/writev()* system calls for normal synchronous I/O, except that *listio()* does not provide the implicit atomicity guarantee of *readv()/writev()*. The application may freely mix read, write, and no-op (ignored) *aiocbs* in the list and may choose to return immediately (*LIO_NOWAIT*), return synchronously when all requests are complete (*LIO_WAIT*), or return immediately but receive an asynchronous event when the entire list is complete (*LIO_ASYNC*).

The *acancel()* function cancels either an explicit *aiocb*, by passing the user address of the *aiocb* used in the initiating *aread()/awrite()/listio()*, or all asynchronous I/O outstanding on a given file descriptor. A successful *acancel()* call returns one of three status indications: (1) all requested operations were canceled, (2) all requested operations were already complete (not *EINPROG*), or (3) at least one of the requested operations could not be canceled.

The *iosuspend()* function is the basic method of synchronization for an application performing asynchronous I/O. This function accepts an array of pointers to *aiocbs* and suspends the calling process until at least one of these requests has completed or until the process receives an event or a signal. The return value of an *iosuspend()* call that returns due to the completion of at least one of the listed requests is the address of the relevant *aiocb*. For multiple simultaneous completions, it is unspecified which completing *aiocb* pointer is returned and the application must scan the list of *aiocb*'s for all completions. Event notification on completion, if requested, is not changed by *iosuspend()*, so, for example, *iosuspend()* may be awakened by an event from a request for which it is not waiting.

```
int      aread(int fildes, struct aiocb *aiocbp); /* asynchronous read */
int      awrite(int fildes, struct aiocb *aiocbp); /* asynchronous write */

/* list directed I/O */
int      listio(int cmd, struct liocb *list, int nent, struct event *event);

/* cancel asynchronous I/O request */
int      acancel(int fildes, struct aiocb *aiocbp);

/* wait for asynchronous I/O completion */
int      iosuspend(int cnt, struct aiocb *aiocbp[]);

/* asynchronous I/O control operation - AIX extension */
int      aioctl(int fildes, struct aiocb *aiocbp);
```

Figure 4: Asynchronous I/O functions

POSIX Implementation Design

The design we chose for our experimental implementation of POSIX asynchronous I/O was influenced by our goals and the operating system base from which we started. Our primary goal was to achieve high performance I/O with the potential to most efficiently use the high I/O bandwidth available in a large System/370 or System/390 configuration. Our target applications were databases and foreign filesystems that would initially require only raw access to disk and tape. Our base operating system was the general release code for AIX v1.2, a merged kernel that has small machine specific portions for System/370 (AIX/370) and i386 (AIX PS/2) architectures. AIX/370 currently executes as a guest virtual machine under the Virtual Machine/Extended Architecture operating system. AIX/370 is based on System V Release 2 (originally IX/370), but it has evolved with additions from System V Release 3 and 4.3BSD. In particular, AIX v1.2 includes the Transparent Computing Facility, which is an instantiation of the LOCUS distributed system architecture [35, 44]. LOCUS provides for a single system image view of a cluster of heterogeneous machines with a distributed, replicated filesystem and migration of executing processes among compatible architectures. The LOCUS filesystem offers some features of a database, including a commit mechanism and optional rollback of file updates [32]. Thus, AIX/370 has an advanced filesystem design, and we decided not to include regular file support in our experimental implementation. The AIX/370 filesystem does not currently include a File System Switch or virtual file system (vnode) layer.

There are at least two general approaches to asynchronous I/O under UNIX. The first involves using a set of kernel processes to block for each outstanding request while releasing the user process to continue. The second involves interfacing directly with the UNIX drivers and making the asynchrony existing in the device strategy routine visible to the user process. Using kernel processes has several advantages: there are no changes required to any driver, the facility will work with any type of file equally well, and there are no changes needed in the virtual file system layer. The disadvantage is that there is still one context switch per request, though this is a kernel process context switch which is somewhat less expensive than a typical user context switch. In general, this is a considerably easier implementation strategy than the alternative.

Interfacing directly to the UNIX drivers has the opposite side of each of the above points. Every driver that will support asynchronous I/O must be changed, though the changes are relatively minor if a strategy routine is already available. A driver level implementation does not immediately work with any type of file other than character special files,

although with extra effort all file types may be supported. Worst of all, asynchronous I/O at the driver level requires restructuring some of the virtual file system operations. The original vnodes paper [20] and more recent work [39] emphasize that vnode operations are assumed to be performed in the context of the calling process. While the OSF virtual filesystem is being extended in various ways [19], the ramifications of asynchronous vnode operations appear to be quite extensive and require careful study. For example, mapped files present a significant challenge.

With all these disadvantages, the one clear advantage of a direct driver interface is efficiency. By loading the strategy routine with many outstanding requests, the calling process may sleep until all operations are complete before requiring any context switches. In addition, support for request priority may require changes to the driver strategy routine. The kernel process design will perform a context switch in some sense for every completed request. A hybrid design for asynchronous I/O might be feasible, with regular files using kernel processes and character special files using direct driver access, although this would appear to have a significant amount of duplicated implementation effort. We chose to include asynchronous I/O by interfacing directly with the device drivers, and the following sections supply details of our implementation.

Events Implementation

The great majority of the events implementation was patterned after the signal implementation existing in AIX/370. AIX signals are reliable, with a flag available for System V signal semantics if desired. In one sense, events are simpler than signals, because signals must deal with dozens of special cases while events always have user defined meanings. This simplicity breaks down when events are grafted onto the existing signal mechanism, and the interaction between events and signals requires careful specification.

AIX/370 supports a maximum of 64 signals (two longwords). For simplicity, our implementation supports NEVT = 64 event classes, numbered from zero, since we were thus able to use many of the stock signal manipulation macros and functions unchanged. Events required several additions to the *proc* and *user* structures, as listed in Figure 1. The *p_evtmask* and *p_evt* elements are analogous to *p_sigmask* and *p_sig* in a typical UNIX. The *p_evthead* and *p_evttail* arrays are the head and tail of the pending event list for each event class, while the *p_sigclass* array maps signal numbers into event classes. These three arrays add noticeably to the size of the *proc* structure, but we feel this is not a problem on larger machines. For smaller machines or where real memory is at a premium (if the process table is not paged), a smaller number of

supported events might be more appropriate.

The *p_evtpoll* element records the set of event classes being polled during the *evtpoll()* system call. The *p_evtpending* element keeps a running total of the number of events pending (queued) for the process. It is used to impose the system policy limits for *evtpending_max*, the total number of queued but blocked events for a given process. Since each event queued but not immediately delivered consumes some system resources, this prevents a runaway process from blocking event delivery and then posting an unlimited number of events. The *evtpending_max* variable is a standard tunable system parameter, with a default of 50. The *user* structure holds various information analogous to the BSD signal stack and system call interrupt masks, though these functions are not currently implemented. Other elements store the saved event mask and timeout limit during an *evtpoll()*.

Both events and asynchronous I/O require similar queuing systems. One of our design goals was to decouple the queue manipulation facilities from the queue element allocation strategy. This allows us to use dynamic kernel memory allocation for now and later to change to a statically allocated list for deterministic response, if needed. The kernel queue processing routines include: *evtalloc()*, which allocates resources for a given event structure and links it at the end of the correct class queue; *evtnext()*, which dequeues and returns queue elements in FIFO order, or NULL, if the queue is empty; and *evtdump()*, which discards all queued events for a given process.

The procedure for posting and delivering an event is not complex. The kernel posts the event internally by calling *pevent()* to manipulate the necessary data structures in the process' *proc* structure. The process being posted need not be in context (i.e., have its virtual memory mapped in), since the relevant structures are all accessible through the process handle and the process table is not paged by AIX/370. The *pevent()* function queues the event and marks the appropriate class as pending in *p_evt*. If the event is unblocked, or blocked and being polled by *evtpoll()*, the process is awakened to handle take the event. The kernel then finishes the system call, interrupt, or whatever prompted the event. The next time the kernel is about to return to user mode for the target process, from a system call, page fault, clock tick, I/O interrupt, etc., the *p_evt()* routine determines if any events are waiting to be delivered to the process. If any unblocked events are queued, *p_evt()* calls *sendevt()* to actually deliver the event.

The *sendevt()* function builds a user stack frame that simulates a call to the event's *evt_handler()* element, stores the user's register, signal, and event context on the stack, and places an *rfe()* (Return From Event) system call as the last

instruction of the event handler sequence. This sequence is the well known "signal trampoline" [23]. Because events are queued, *p_evt()* builds the event stack for the highest priority event waiting to be delivered. All undelivered events will eventually be handled as the *rfe()* call returns to user mode through *p_evt()*. The semantics of *evtpoll()* require that the process in an unsatisfied poll remain suspended after the execution of unblocked event handlers. Thus, when all handlers have been run and the system would normally return to user mode, if the process is still in an unsatisfied *evtpoll()*, the user return address is adjusted to restart the poll. The draft is silent on the semantics of nested polls within event handlers while already polling in *evtpoll()*. We return an error on the second poll attempt in this odd case.

One of the goals of the draft events facility was to minimize changes to the existing signal implementations. As a practical matter, we were able to delay the final merging of events and signals until quite late in our development. By keeping separate calls to *p_evt()* and *p_sig()* for building event and signal handler stacks, respectively, we ignored the combined semantics of signals and events but, in return, we did not have to modify the signal code in any way while we were debugging most of the events implementation. By leaving signals unqueued, the final merge is simple and isolated in its effect on the vast majority of the signal code.

UNIX I/O Operation Paths

Normal Synchronous I/O

To understand the changes necessary to support asynchronous I/O, it is instructive to trace the path of a normal synchronous I/O operation. There are two ways to initiate an I/O in a full-blown UNIX driver: by a raw read/write or by a block read/write [2, 3]. The raw read/write starts with a *read()/write()* system call, that eventually calls the driver's *ddread()/ddwrite()* routine directly and transfers data to/from the user buffer without an intermediate copy in a kernel buffer. The block read/write uses the kernel buffer cache to block and unblock the data before calling the driver *ddstrategy()* routine. The strategy routine maintains a queue of work awaiting execution and, in the optimal case, can start the next I/O operation inside the driver interrupt routine. The *ddread()/ddwrite()* routines can either start the I/O directly (as in some simple tape drivers, for example) or can package the request in a *buf* structure and call the strategy routine to queue it (as in most disk drivers). For disks, multiple independent processes may be stacking requests, and this requires a queue and a strategy routine to manage it. The strategy routine is named for its ability to reorder the outstanding requests to optimize overall system throughput, as in disk arm and rotational scheduling.

Let us start with a raw character device read, *read(fildes, buffer, count)* for example. First, the read system call traps into the operating system and the parameters are checked for validity. The file type (character, block, regular file, FIFO, directory, etc.) determines how the file is read. In our case (character file), the file offset is set and then *readi()* is called. The *readi()* function reads the file data from the given inode at the current offset. The device number from the inode is used to index into the device switch table and the appropriate *ddread(dev)* routine is called. The various parameters used by *ddread()*, such as file offset, user buffer address, and number of bytes to read, are passed implicitly in the calling process' u-area (user area). The driver then either packages the call into a *buf* structure and calls *ddstrategy()* to queue the operation for later execution or starts the I/O directly.

The standard routine provided in Unix to package I/O requests from *ddread()/ddwrite()* and pass them to the *ddstrategy()* entry point is *physio()*[1]. The *physio()* routine takes as parameters a strategy routine, a *buf* structure to be filled, the major and minor device numbers, a flag indicating read or write, and (in AIX) a pointer to a routine to check for various block size conditions imposed by the driver and device hardware. For example, individual transfers must be multiples of a 4K block, start on a block boundary, and be no more than 60K for the current disk driver. The *physio()* routine copies the u-area parameters into the *buf* structure and then enters a loop that performs transfers up to the maximum chunk size (60K for disk) by page-locking the relevant pages of the user's buffer in real storage, calling the strategy routine to queue the physical transfer, and sleeping until the request is complete. If any of the buffer pages are not resident (invalid pages), they are accessed in sequence to bring them into real memory. The driver interrupt routine calls *iodone()* when the transfer is complete, and *iodone()* in turn awakens the user process asleep in *physio()*. The *physio()* routine unlocks the pages that had been locked down, updates the u-area state, and loops if there are more data to transfer. Since *physio()* sleeps while the request is on the strategy queue, the requesting process is blocked in the *read()* system call, only returning upon the completion of the request. When the driver *ddread()* call returns, the *read()* system call updates the file position for the file affected and returns from the system call to the user's process.

Asynchronous I/O Path

An *aread()/awrite()* system call follows much the same path as above, with an extra asynchronous I/O control block (*aiocb*) pointer passed in the original call. This *aiocb* is copied into kernel space, extra kernel state is added, and it is placed on various queues. The system call proceeds as above,

with parallel routine names *aread()*, *areadi()*, and *ddaread()*, until the driver would normally call *physio()*. Because the user process cannot be allowed to sleep in *physio()*, its functions have been reorganized into two pieces, a "top" half, *aphysio_top()*, that executes in the context of the user process (with a valid u-area) and a "bottom" half, *aphysio_bot()*, that executes in interrupt context, where no specific process can be assumed to be in context. The top half retrieves the u-area parameters it needs and stores them in the *aiocb*, checks that the request is a proper multiple for physical I/O, locks the first set of buffer pages in physical memory (page faulting as necessary), marks the *aiocb* as active with *EINPROG*, calls the strategy routine to queue the physical transfer, and returns to the driver *ddread()/ddwrite()*, which returns immediately through *areadi()* and *aread()* to the user process.

When the strategy request is complete, the unchanged driver interrupt routine calls *iodone()*, which has a new check for an asynchronous operation and calls *aphysio_bot()* to continue or complete the transfer⁴. Then *aphysio_bot()* unlocks the pages that had previously been locked, updates the state of the transfer which is completely stored in the *aiocb* (since the current u-area has the context of some random process), and loops until there are no more data to transfer. If the request is not complete, *aphysio_bot()* page-locks the next set of buffer pages and again calls the strategy routine for the next portion of the transfer.

When the entire operation is complete, *aphysio_bot()* need only update the values in the user's *aio_errno* and *aio_nbytes* and perform final processing on the queue element. The AIX/370 kernel does not currently support writing into an arbitrary real memory location in interrupt context, and manipulating segment and page tables in interrupt context is not attractive. Instead, we arranged for *aphysio_bot()* to queue a request to update the *aio_errno* and *aio_nbytes* members of the user *aiocb*, in the user's virtual address space. Since *aphysio_bot()* is operating in interrupt context, the *aio_errno* posting is done the next time the user process returns to user mode from kernel mode, at the same time as signals and events are posted. This is "instantaneous" I/O completion notification on all uniprocessor configurations. If the requesting process was executing when the interrupt occurred, it will see the *aio_errno* value updated before the interrupt trap returns to that process. If the requesting process was not executing, it will see the new *aio_errno* before it ever again runs. On a System/370 multiprocessor configuration, this can cause problems when processor A requests I/O but processor B takes the I/O interrupt. If the requesting process is

⁴*iodone()* still performs a *wakeup()*, but no process sleeps on completion of an asynchronous transfer.

compute-bound and does no system calls, it might not be notified of I/O completion for up to one clock tick. Finally, *aphysio_bot()* wakes up any processes sleeping on I/O completion, such as *iosuspend()* and certain forms of *listio()*, and posts any requested event to the user process.

For existing drivers that have a *ddstrategy()* routine, entry points for *aread()*, *awrite()*, *acancel()*, and *aioctl()* are easily added. These new routines call *aphysio_top()* instead of *physio()*. No changes to the strategy routine were immediately necessary. Support for asynchronous I/O priority will require modifications to the algorithms that sort queued requests in the strategy routine. The *acancel()* function should search the strategy queue and remove any buffers that have not yet started, and, if possible, issue commands to stop data transfer on the currently executing buffer. This allows *close()*, *_exit()*, *shmdt()*, etc. to continue safely rather than wait on asynchronous request completion. In a minimal implementation, *acancel()* may always return failure, causing these routines to wait as necessary for all requests to complete. Old drivers are completely upwardly compatible, but currently must be recompiled to track changes to the *proc* and *user* structures.

Asynchronous I/O Implementation

Asynchronous I/O required several additions to the *proc* structure. In Figure 3, *p_aioqueued* is the total of all asynchronous requests on any queue, *p_aioactive* is the total number of requests active on all file descriptors, and *p_aiodone* is the total of requests awaiting *aio_errno* posting (see below). The *p_aio* element is the head of the *aio_errno* post list. The *p_aiopagelock* member is the total number of pages locked in physical memory with active asynchronous I/O, which may not exceed a system tunable parameter, *aiopagelock_procmax*, to prevent a single process from consuming too much physical memory. Another system tunable parameter, *aiopagelock_sysmax*, limits the total number of physical pages that may be locked down for asynchronous I/O systemwide and prevents memory deadlock. Pages to be locked down are allocated early in asynchronous calls and reserved throughout the active request.

The *p_liocount* member tracks the number of active requests for a process that executed a *listio()* call and is now LIO_WAITing for all requests to complete so it may resume execution. The *file* structure, in *<sys/file.h>*, includes two new elements: *f_aio*, which is the head of the active request list or NULL, if no asynchronous I/O is active on that file, and *f_aioactive*, which is a count of that file's active requests. Each request is always inserted at the front of the file descriptor queue, but the order in which requests are executed is determined completely by the driver strategy routine, possibly influenced by the

aio_prio priority parameter. The *buf* structure, in *<sys/buf.h>*, includes one new element, a pointer to its associated control block, *b_aiocb*. The typical request will then have an *aiocb* that includes a *buf* structure, *aio_bufhead*, and *aio_bufhead.b_aiocb* points to the *aiocb*. This linkage is necessary because driver strategy routines accept pointers to *buf* structures, not *aiocb*s.

The basic kernel queue element for asynchronous I/O in our implementation is a *kaiocb* structure [Figure 3]. The user space *aiocb* structure is composed of the first eleven members of this structure. This *aiocb* structure deviates slightly from the draft by moving the requested number of bytes and the user buffer address into the *aiocb*, as suggested by the Common Reference Ballot [6]. That is, we have implemented the syntax *aread(int fd, struct aiocb *aiocbp)* rather than *aread(int fd, char *buf, int nbytes, struct aiocb *aiocbp)* as specified in the draft. Corresponding adjustments were made in the *liocb* structure of *listio()* arguments. The draft rational states that its syntax of arguments was chosen to have the least number of changes to the POSIX.1 standard [17]. An odd semantic feature of this syntax calls for a NULL *aiocbp* to perform an asynchronous operation at the current file offset, but no status is returned and no event notification is given upon completion. While this certainly gives a lower overhead read or write, we find the absence of any synchronization or error information to offer unacceptable nondeterminism in a system function. A "read maybe" facility may not be useful for input, unless the data is uniquely self-describing. On output, this "write maybe" function might be used for extremely low priority log files, but the application can never be sure when it may reuse the output buffer.

The *aio_ext* and *aio_mpxchan* elements of *kaiocb* are specific to AIX. The *aio_ext* parameter, being an optional parameter input to the system from the application, requires a corresponding validity flag in the *aio_flag* argument according to the latest revision of the POSIX.1 standard. Otherwise the system might not be able to distinguish an uninitialized *aio_ext* value from intended input. Most of the other members of the *kaiocb* structure are storage for state that must be saved for use when the requesting process is out of context.

The kernel asynchronous I/O queue manipulation routines include: *aioenqueue()* and *aiodequeue()*, which enqueue and dequeue kernel *aiocb*s to and from various queues and update the corresponding counts; *aioalloc()*, which allocates resources for a given *aiocb* structure and links it onto the end of the file pointer list; *aioshift()*, which shifts an element from an active file descriptor list to the process *aio_errno* post list; *aiofree()*, which removes an element from the list and frees its resources; and *aioflush()*, which attempts to cancel

all requests active for a given process and file descriptor. If *aioflush()* is called by *close()*, *exit()*, *shmdt()*, etc., it blocks waiting for any uncancelable operations. The semantics of *acancel()* require *aioflush()* to return immediately even if not all requests were canceled. The enqueue function places requests on a circular list in time order. To flush the queue, *aioflush()* scans the queue from youngest to oldest to optimize the chances of successfully canceling newer requests before they become active in the driver. To validate and queue an *aio*cb, *aio_validate()* checks for user access to the *aio*cb and legal values for the *aio_prio* and, if a completion event is requested, the *aio_event.evt_class* parameters. The number of page frames to be locked is calculated and reserved.

The basic *aread()/awrite()* functionality required only minor changes to the existing kernel *read()/write()* code, including removal of code for unsupported file types in our experimental implementation and changing a few routines to return error numbers through *aio_errno* rather than the process' *u.u_error* variable. We were able to share much of the asynchronous read/write code between *aread()/awrite()* and the *listio()* read/write processing. The LIO_WAIT and LIO_ASYNC options of *listio()* require maintaining extra state. As *aphysio_bot()* completes each LIO_WAIT request, flagged as such in *aio_flag*, it decrements *p_liocount* and when this counter reaches zero, the process is awakened. For LIO_ASYNC, a separate structure is maintained that counts down the outstanding requests and posts the requested event on list completion.

The *iosuspend()* system call presented some interesting problems. According to the draft resolution of issues [18], the POSIX networking group has tentatively agreed to use asynchronous I/O and *iosuspend()* in place of standardizing a *select()* function and non-blocking I/O. As pointed out in the Common Reference Ballot [6], having the application process refer to previously issued operations by passing a user address is unprecedented in UNIX. In the past the system has accepted a user request and returned a unique system-generated handle. An obvious implementation strategy for *iosuspend()* is to find the kernel address of the queue element that corresponds to the argument user virtual address of the user's *aio*cb and flag each such queue element so, on completion, it awakens the *iosuspend()*ed process. But the argument user virtual address might correspond to any (or multiple) queue element on one of dozens of open file descriptors or on the *aio_errno* post queue, or it might have already completed. We considered various data structures but finally decided to experiment with a simple, if unorthodox, solution: we always write the kernel virtual address of the *kaiocb* queue element back into the user's copy of his *aio*cb. Within the AIX/370 memory architecture, user and kernel addresses share

the same virtual address space, so such an operation is relatively inexpensive. By validating the kernel virtual address retrieved from user space as a correct kernel address and checking for correct process handle, progress indicator (EINPROG), and magic number, we can, in constant time, determine the kernel queue element to tag as involved in an *iosuspend()*. Of course, this is different from normal UNIX practice and could have unacceptable security implications. After determining the address of the kernel queue element, the *aio_suspaddr* element is set to the address of an integer, initially -1, and the *aio_suspindex* receives the index in the user's array of *aio*cb pointers for a return value. For the first tagged request that completes, *aphysio_bot()* sets the integer at *aio_suspaddr* to the value in *aio_suspindex*, awakens the process suspended by *iosuspend()*, and the index value is returned to the application.

Latent Page Faults in Interrupt Context

Memory locking is handled by several special asynchronous routines. Similar to other standard routines, *aphysiolock()* and *aphysiounlock()* lock and unlock all pages in a given virtual address range. The *aphysiolock()* routine is designed to be called in the context of the process whose memory is being locked [i.e., *aphysio_top()*] as it accesses each page to cause it to be paged in, if necessary, while *aphysiounlock()* accepts a process handle and can unlock the pages of an arbitrary process. The *aphysiolocknw()* routine attempts to lock the pages of an arbitrary process, not necessarily in context, but returns failure with all lock counts unchanged if it is unable to lock the entire virtual address range requested. By accepting a process handle of a process guaranteed to have its page tables resident⁵, *aphysiolocknw()* may safely be called in interrupt context by *aphysio_bot()* to try to lock the next set of physical pages for a transfer. If locking fails in interrupt context because the necessary pages are not currently resident, how should this be dealt with? One solution is to arbitrarily limit the maximum transfer size for any request to what can be conveniently locked at one time. For realtime processes, this is certainly a valid approach, since handling page faults during high priority operations is unacceptable. But for general high performance I/O, we find such a restriction very limiting. There is no intrinsic limit on the size of a synchronous *read()* or *write()*, and page faulting occurs behind the scenes as necessary to handle a large request. We did not want to force applications to change the natural size of their data requests to match an arbitrary limit for physical I/O.

⁵In AIX/370, the process flag SPHYSIO ensures that the process is ineligible to be swapped out

If *aphysio_bot()* detects a latent page fault in interrupt context, it moves the partially completed request to a special queue and awakens a kernel daemon process that takes the necessary steps to make the pages resident and calls the driver strategy routine again to continue the transfer. This requeuing reorders the requests and is therefore only feasible for random access media, such as disk. Sequential media, such as tape, must not have their requests reordered. The asynchronous I/O page fault daemon process calls *apagein()*, a modified version of the *pagein()* system routine, that brings pages from disk into physical memory. Normally, *pagein()* is called indirectly from the system trap code after a page fault exception, while the process needing the page is in context. By supplying the process handle as an argument to *apagein()*, we are able to call the pager directly. The asynchronous I/O page fault daemon blocks during each page fault, so on a very busy system multiple daemons may be required. It appears that future versions of BSD will support a similar kernel facility to perform a page-in operation for other than the currently running process [29].

Asynchronous *ioctl()* Operations

The POSIX.1 standard declined to standardize the ubiquitous UNIX *ioctl()* function. The *ioctl()* function is a general mechanism for performing device and file specific operations which are often highly nonportable. The draft also declines to standardize any form of *aiocb()*, leaving vendors to implement whatever form they see fit. Since we already had the machinery in place for handling asynchronous requests, and it was necessary to restructure the existing AIX/370 tape driver to include a strategy routine, we decided to implement an *aiocb()* function for tape. This allows a process to give a sequence of read/write and rewind/space/unload commands and, using *listio()*, receive one notification on completion of the entire set of operations. Any unrecoverable error will flush the entire tape driver strategy queue because proper media positioning is implicit in later requests. This facility is almost purely for programming convenience, as the performance gains from restarting I/O immediately upon the drive achieving the desired position are minimal. The advantage derived from this facility is the uniform program structure allowed when all I/O completions can generate asynchronous notifications.

An *aiocb()* operation accepts two arguments [Figure 4]: a file descriptor and a pointer to an asynchronous I/O control block. The normal parameters of an *ioctl()* are passed by overloading unused elements of the *aiocb* [Figure 3]. Because the calling process is not necessarily in context when the *aiocb()* is executed, any data transfer into or out of the kernel must be staged in a kernel buffer and handled when the process returns to context. Such a

mechanism is already implemented in AIX, where such *ioctl()*s are termed "well-behaved" [16]. Originally from BSD UNIX, well-behaved *ioctl()* routines have their commands defined through macros that encode the direction of data movement and the number of bytes transferred in the upper bits of the command. The driver's *ddioctl()* and *ddaiocb()* routines may then use direct copy methods, such as *bcopy()*, to and from a kernel buffer area, instead of indirect copy methods, such as *copyin()* and *copyout()*, that move data between user and kernel address spaces. All *aiocb()* commands must be well-behaved.

Asynchronous I/O Issues

The semantics of asynchronous I/O may have some interesting security implications. In a trusted computer system, processes and files are tagged with information labels that float upward as the relevant objects touch more secure objects. For a normal, synchronous I/O operation, the labels are adjusted appropriately at the conclusion of the I/O operation. Because asynchronous I/O operations return control immediately, the opportunity exists for mixing of levels if the levels are not floated at initiation. The apparent solution is to float the labels at both initiation and completion, which imposes additional overhead on a secure operating system. In addition, the undefined behavior of shared *aiocbs*, such as the order of both a read and write, must nevertheless be guaranteed to produce secure results.

The draft standard is silent on the final execution order of asynchronous requests, other than requiring that they be documented by the implementation. Write requests are of most interest, as many database and filesystem operations require a facility to sequence physical writes for error recovery. Asynchronous writes are tagged with the *aioprio* argument, but support for priorities is device dependent. Writes might be reordered within a given priority level, or arbitrarily if priority is not supported. These unspecified write ordering semantics are part of the general problem of determining the order of writes to stable storage. The latest intelligent disk controllers have large caches between the system and the media, and can reorder requests arbitrarily. A power failure can lead to serious filesystem corruption if the system had carefully ordered its writes for consistency and hardening, but the controller subsequently "optimized" the physical order. In our implementation, queueing a latent page fault to the asynchronous I/O page-in daemon changes the order of requests on the strategy queue. Only the disk is affected by this requeuing, as the current tape driver limits the physical block size to 64K-1 bytes and locks down all necessary memory at initiation. Sequential media, of course, must not have their I/O requests reordered.

Another feature for possible standardization or vendor extension is a physical sync bit. The current FSYNC facility synchronously waits until the write request has left the system. Again, with intelligent controllers, there may be some additional delay before the request is actually on stable storage. For an asynchronous write, the opportunity exists to simply delay notification until the desired level of safety has been reached. This extra step is virtually costless with asynchronous I/O, since computations and other I/O may proceed apace.

Because *select()* is very common in existing code, the problem of handling the interactions of *select()* with events and asynchronous I/O must be dealt with. A typical situation is for a process to unblock a set of events while waiting for asynchronous I/O on disk and tape and then to call *select()* on a set of network file descriptors. Once in the *select()*, an event will unblock the process with EINTR, but there is a small window during which the event may be delivered before the *select()* starts, and the *select()* could block indefinitely. We are experimenting with an *evtselect()* system call that atomically sets the event classmask and calls *select()*. One obvious long-term approach is to extend all socket system calls to accept *aiocb* arguments for full asynchronous operation. This we have not yet attempted.

Why Not Use Threads?

The question often arises as to why asynchronous I/O is needed at all if lightweight threads are available. If the threads are truly lightweight, with relatively low context switching overhead, then threads can be an effective method of performing non-blocking I/O [26]. A given process would spawn as many threads as needed (maybe three or four) to load a driver's strategy routine so a new operation could begin during the interrupt processing, without having to cycle through a wakeup and rescheduling of the next thread before I/O is restarted. But threads have several disadvantages for high performance I/O. Threads still require one system call per I/O, with no facility to bundle a large number of requests at once as with *listio()*. Threads need explicit synchronization with some master thread controlling many drone threads through shared memory or whatever. This synchronization overhead and complexity is eliminated with true asynchronous I/O, since the application can choose to poll the status values, post individual or group completion events, or explicitly synchronize by way of *iosuspend()*, as necessary. Finally, for sequential media, where the order of reading or writing is crucial, sequencing of asynchronous requests is automatic and implicit. There is a significant programming overhead in managing a large number of threads and arranging for each to block in a definite sequence.

Status and Conclusions

Before and after data measuring the practical effectiveness of asynchronous I/O is required. At this writing, plans are in place for extensive full scale system tests of a real system running real I/O intensive applications. It is certainly easy to generate impressive (10-20X) speedups using asynchronous I/O by comparing against a simple-minded single-threaded sequential application that is I/O bound [28]. For UNIX, the useful comparison is against a real application using an effective substitute for asynchronous I/O, such as a master I/O process with many drones. In addition, asynchronous I/O does not make the I/O devices or subsystems themselves run any faster. There is a higher probability of being able to start a new operation in interrupt context, instead of cycling through sleep/wakeup for the next transfer, but the main saving is in processor cycles. In UNIX, the test of an efficient asynchronous I/O implementation is its effect on total system throughput by freeing more central processor resources for useful work instead of consuming cycles in context switching.

We have discussed many of the general issues involved in implementing any asynchronous I/O facility in UNIX, with specific details of our experimental implementation of the unapproved POSIX draft. The general availability of asynchronous I/O at the user process level is a new feature of UNIX as it evolves to meet application requirements. Programming paradigms will change as applications switch from synchronous or even non-blocking I/O to fully asynchronous I/O. True asynchronous I/O facilities offer the promise of simpler application structures, where the main line of an application may initiate I/O requests, and completions and errors are handled asynchronously as necessary.

We wish to acknowledge our debt to the fine work by the dedicated members of the P1003.4 committee. Many thanks are due Howard Green, Wally Iimura, Win Bo, Doug Locke, Paul Davis, Kathy Bohrer, Rich Ruef, and Steve Kiser for their support and assistance at various stages of this work.

References

- [1] AT&T. *UNIX System V and V/386, Release 3, Block and Character Interface (BCI) Driver Reference Manual*, 1988. Select Code 307-192, Issue 2, Preliminary.
- [2] AT&T, Lisle, IL. *UNIX System V and V/386, Release 3, Block and Character Interface (BCI) Driver Development Guide*, 1989. Select Code 307-191, Issue 2.
- [3] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Englewood Cliffs, NJ, 1986.
- [4] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, Englewood

- Cliffs, NJ, 1977.
- [5] Cray Research, Inc., Mendota Heights, MN. *Volume 4: UNICOS System Calls Reference Manual*, 1989. SR-2012 5.0.
 - [6] P1003.4 Draft 9 Common Reference Ballot, 1990. comp.std.unix, 19(124), May.
 - [7] Digital Equipment Corp., Maynard, MA. *VMS System Services Reference Manual*, April 1988. Version 5.0, #AA-LA69A-TE.
 - [8] Digital Equipment Corp., Maynard, MA. *ULTRIX Reference Pages, Section 4: Special Files*, June 1990. Version 4.0, #AA-LY17B-TE.
 - [9] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43-112. Academic Press, 1968.
 - [10] Janet I. Egan and Thomas J. Teixeira. *Writing a UNIX Device Driver*. Wiley, New York, 1988.
 - [11] Donald E. Freeman. *I/O Design: Data Management in Operating Systems*. Hayden Book Company, 1977.
 - [12] Borko Furht, John Parker, Dan Grostick, Hagai Ohel, Tom Kapish, Ted Zuccarelli, and Orlando Perdomo. Performance of REAL/IX - fully preemptive realtime UNIX. *ACM Operating Systems Review*, 23(4):45-52, October 1989.
 - [13] Mark Gary. Overcoming UNIX kernel deficiencies in a portable, distributed storage system. In *Crisis in Mass Storage: The Tenth IEEE Symposium on Mass Storage Systems*, pages 134-139, Washington, DC, May 1990. IEEE Computer Society Press.
 - [14] Harris Computer Systems, Ft. Lauderdale, FL. *CX/UX Programmer's Reference Manual, Volume 1*, October 1990. 0890111-050.
 - [15] IBM, San Jose, CA. *MVS/Extended Architecture System-Data Administration*, 1987. GC26-4010.
 - [16] IBM, Danbury, CT. *AIX Operating System Technical Reference Volume 2*, 1989. Version 1.2, SC23-2301-00.
 - [17] IEEE, editor. *IEEE Standard Portable Operating System Interface for Computer Environments*. 1988. ANSI/IEEE Std. 1003.1-1988.
 - [18] IEEE, editor. *Realtime Extension for Portable Operating Systems*. December 1, 1989. Preliminary Unapproved Draft P1003.4/D9.
 - [19] Michael L. Kazar, Bruce W. Leverett, and Owen T. Anderson. DEcorum file system architectural overview. In *Proceedings of the Summer USENIX Conference*, pages 151-163, Anaheim, CA, June 11-15, 1990.
 - [20] Steven R. Kleiman. Vnodes: An architecture for multiple file system types in SunUNIX. In *Proceedings of the Summer USENIX Conference*, Atlanta, GA, 1986.
 - [21] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, pages 211-227. Addison-Wesley, 2nd edition, 1973.
 - [22] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, pages 320-378. Addison-Wesley, 1973.
 - [23] S. J. Leffler, M. K. McKusick, and M. J. Karels. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.
 - [24] Alan L. Leiner. System specifications for the DYSEAC. *JACM*, 1:57-81, 1954.
 - [25] Stuart E. Madnick. *Operating Systems*, pages 52-104. McGraw-Hill, New York, 1974.
 - [26] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating System Principles*, pages 191-201. ACM, December 3-6, 1989.
 - [27] MASSCOMP, Westford, MA. *UNIX Programmer's Manual, Volume 1B*, February 1988. Revision K, order M-RTU-011.
 - [28] Michael Mayfield. Improving performance for disk-bound programs. *DEC Professional*, 8(2):86-92, February 1, 1989.
 - [29] M. K. McKusick, M. J. Karels, and K. Bostic. A pageable memory-based filesystem. In *Proceedings of the Summer USENIX Conference*, page 139, Anaheim, CA, June 11-15, 1990.
 - [30] Frederick C. Mish, editor. *Webster's Dictionary*. Merriam-Webster, Springfield, MA, 1987.
 - [31] Modular Computer Systems, Inc., Ft. Lauderdale, FL. *REAL/IX Operating System Concepts and Characteristics*, 97xx Systems, June 1989. #205-855001-000.
 - [32] Erik T. Mueller, Johanna D. Moore, and Gerald J. Popek. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 71-89. ACM, Oct 10-13, 1983.
 - [33] Object Design, Inc., Burlington, MA. *Object-Store Technical Overview*, 1990. Release 1.0.
 - [34] Charles S. Peskin and David M. McQueen. 3-dimensional fluid dynamics in 2-dimensional central memory: Vectorization and asynchronous I/O. In Robert B. Wilhelmson, editor, *High-Speed Computing: Scientific Applications and Algorithm Design*, pages 72-73. University of Illinois Press, 1988.
 - [35] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture*. MIT Press, Cambridge, MA, 1985.
 - [36] Pyramid Technology Corporation, Mountain View, CA. *OSx 4BSD Manual Pages, Volume II*, May 1990. 4100-0046-D.
 - [37] Marc J. Rochkind. *Advanced UNIX*

Programming. Prentice-Hall, Englewood Cliffs, NJ, 1985.

- [38] Saul Rosen. Electronic computers: A historical survey. *ACM Computing Surveys*, 1(1):15, March 1969.
- [39] David S. H. Rosenthal. Evolving the vnode interface. In *Proceedings of the Summer USENIX Conference*, page 139, Anaheim, CA, June 11-15, 1990.
- [40] Louis D. Stevens. Engineering organization of input and output for the IBM 701 electronic data-processing machine. In *Review of Input and Output Equipment Used in Computing Systems*, page 81. Joint AIEE-IRE-ACM Computer Conference, December 1952.
- [41] W. Richard Stevens. *UNIX Network Programming*, pages 326-331, 389-391. Prentice-Hall, Englewood Cliffs, NJ, 1990.
- [42] Sun Microsystems, Inc., Mountain View, CA. *SunOS Reference Manual, Vol. II*, 1990. Release 4.1, #800-3827-10.
- [43] A. J. van de Goor. *Computer Architecture and Design*, pages 305-328. Addison-Wesley, Reading, MA, 1989.
- [44] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 49-70. ACM, Oct 10-13, 1983.
- [45] H. Wettstein and G. Merbeth. The concept of asynchronization. *ACM Operating Systems Review*, 14(4):50-70, October 1980.
- [46] R. J. Wicks. Balanced systems and capacity planning. Technical report, IBM Washington Systems Center, 1988. Technical Bulletin GG22-9299-02.

Lester Buck specializes in systems integration and filesystems for various Fortune 500 clients. He is a member of ACM and the IEEE Computer Society, and a former member of ANSI X3B11.1, WORM Volume and File Structure Standards. He received a B.S. in Physics from Caltech in 1975 and a Ph.D. in Physics from the University of Illinois, Urbana, in 1982. His electronic address is buck@siswat.lonestar.org.



Robert Coyne is a Senior Systems Engineer in IBM's Federal Sector Division (FSD). He is responsible for High Performance Data Systems research and development at IBM FSD Houston. His research interests include multi-vendor interoperability and very high speed data distribution and filesystems. He received a B.S. in Computer Science from the University of Texas at El Paso in 1979 and an M.S. in Computer Science from the University of Houston at Clear Lake in 1983. Use coyne@houvmscc.iinus1.ibm.com to reach him electronically.



The Parallelization of UNIX System V Release 4.0

*Mark Campbell, Richard Barton, Jim Browning, Dennis Cervenka,
Ben Curry, Todd Davis, Tracy Edmonds, Russ Holt, John Slice,
Tucker Smith, Rich Wescott* – NCR Corporation—E&M Columbia

ABSTRACT

This document describes the fine-grained parallelization of the UNIX System V Release 4.0 kernel for a tightly-coupled symmetric multiprocessing machine. Unlike most multithreading efforts performed using an existing single-threaded UNIX base, this effort focused on altering as little source as possible during the initial port and then making algorithmic modifications only when a measured performance problem manifested itself. This multiprocessing kernel (SVR4/MP) was designed, implemented, and tuned in a 9 month development cycle and met all of its scalability goals in a dyadic configuration.

Overall design strategy and locking primitives are discussed in this paper from a technical and schedule perspective; in addition, private data, and interprocessor communication are examined. The multithreading strategy for each of several major subsystems is also examined. All of this is followed by a performance and scalability analysis of the overall kernel for several different types of benchmarks on a prototype dyadic MC88000-based system.

Introduction

In June of 1989 NCR established a plan to become one of the first to implement a scalable multiprocessing UNIX System V Release 4.0 (SVR4/MP) kernel with fine-grained locking in all kernel subsystems. We decided that in order to accomplish this we had to have a stable kernel ready for customer availability by March 1990 — a development cycle of approximately 9 months. NCR had already begun a port of the first SVR4 early access release (3B2 load K13) to one of its uniprocessing TOWER family members; this code and the subsequent early access release (K14) was taken and used in an evaluation of personnel and other resources necessary to complete a fine-grained multiprocessing kernel quickly while achieving effective scalability. The SVR4 kernel was broken down into 52 subsystems with the criteria for designation as a subsystem being solely the amount of interdependencies existing between a subsystem and any other subsystem.

Early in the evaluation we found that we had to adopt several constraints on this effort:

- Incremental development of SVR4/MP at the subsystem level throughout the development cycle was initially sought; however, we found no clean way to do this that didn't significantly interfere with normal kernel subsystem interaction. Global locks were deemed much too intrusive; due to the interaction between major subsystems, subsystem locks were also deemed to be too intrusive. We concentrated instead on using a

hierarchical lock debug strategy which allowed us to detect potential problems in both a uniprocessing and multiprocessing environment.

- The amount of interdependence between designated subsystems was great enough that we couldn't scale the number of developers working on the project very effectively — we determined the optimum to be approximately 20 developers. We structured this such that NFS¹ and the UFS filesystem were multithreaded off-site in a staggered fashion; the result was that we had a total of 14 developers on-site.
- It should be noted that the requirement that we have independent parallel development of subsystems by 14 developers greatly influenced the design of the kernel. A great deal of attention was focused upon mechanisms by which coupling between subsystems could be reduced. An example of this was the use of a lock stack mechanism by which the state of the locks held by a process was kept as part of that process's context. Another example was the use of a new set of locking primitives generically termed *Advisable Processor Locks* which allowed developers to concentrate to a large degree on isolated sections of code. In addition, we were able to use and to enhance an extremely

¹RFS was not multithreaded because it was not to be offered in the product.

powerful set of automatic and programmable tools to characterize nested lock interaction.

- The magnitude of the changes between the uniprocessing UNIX System V.3 and V.4 kernels was such that very little existing NCR UNIX System V.2/V.3 multiprocessing kernel code could be used. In addition, the architecture/design differences between our V.2/V.3 multiprocessing subsystems and the AT&T V.4 subsystems were such that little architecture/design was applicable. One major exception to this was the hierarchical lock debug strategy used in earlier NCR products — this was adopted and enhanced, allowing us to quickly get the kernel stable enough to begin scalability tuning.
- The number of differences between K13 and K14 forced us to conclude that we would be forced to undergo rather major integration cycles each time we received an early access load tape. For this reason, as well as past experience with changing large parts of the AT&T kernel source, we decided to do as little algorithmic modification as possible. This strategy served us quite well — the number of differences in the kernel between early access releases K13/K14, K14/K16, and K16/K18 were significant.

This constraint was one of several issues that eventually forced us in most instances to abandon the traditional simple spin and *PSEMA()/VSEMA()* locking used in most multiprocessing kernel implementations (including past NCR multiprocessing kernels). Instead a new type of locking, termed *processor locks*, was used as the primary locking strategy. These locks are discussed in detail in the next section.

- We had found in the past that developers with years of multiprocessing experience still tend to disagree on qualitative lock placement. For this reason a “quantitative versus qualitative” approach to lock placement was chosen in which we concentrated on performing somewhat coarser locking than we had done in the past with a corresponding emphasis on lock debug and contention/performance tools.² We concentrated on bringing up the kernel quickly, performing early scalability tuning with relatively simple benchmarks (e.g., proprietary scalability benchmarks NCR developed to tune multiprocessor systems, the Neal Nelson Business Benchmark, AIM 2.0/3.0, NCR’s System Characterization Benchmark, etc.) and bringing third-party vendors in to allow us to tune with respect to scalability for more complex benchmarks (e.g., TP1, customer case studies, etc.).

²Even with this constraint the developers tended to do

- Concentrate on major scalability improvements; postpone minor improvements until there are no more major improvements. The example most often used was the “mount(2)” system call path — the scalability of this system call was not of overriding importance in this project.

Most of these constraints can be summed up as “Use SVR4 code as is; don’t fix it if it isn’t provably broken badly”. The term “provable” was defined as qualitatively provable to three or more developers or quantitatively provable. The term “badly” was defined as one of the worst problems known to the developer group as a whole. There was quite a bit of concern initially that this approach, while allowing us to make early milestones in our schedule, would tend to overload the schedule towards the end of the development cycle due to the difficulty in tuning the scalability of the kernel. This did not turn out to be the case.

The actual development cycle appeared as in Figure 1.

Milestone	Date
Code Analysis/Uniprocessor Port Start	05/89
Multiprocessing Design Start	06/89
Multiprocessing Code Start	07/89
Multiprocessing Code Complete	09/89
Multiprocessing Kernel To Uniprocessor Prompt	10/89
Multiprocessing Kernel To Multiprocessor Prompt	11/89
Simple Benchmark Scalability Tuning Start	11/89
Complex Benchmark Scalability Tuning Start	12/89
Dyadic Scalability Goals Achieved	02/90

Figure 1: Development Cycle

We believe that minimal code modification, hierarchical lock debug tools, an emphasis on quantitative lock placement, and a tremendous work ethic on the parts of the developers were the major reasons for our success. This schedule is especially impressive when the work that was necessary to track early access releases from AT&T is considered — during this period we received four such releases.

The remainder of this document describes some of the primitives used in SVR4/MP, the design of selected major SVR4/MP subsystems, and a performance analysis of SVR4/MP as implemented on a tightly-coupled dyadic MC88000-based prototype machine. It is assumed that the reader is familiar with basic multiprocessing concepts and SVR4 internals.

fine-grained locking — several problems have been resolved in which we had to make lock granularity more coarse in order to improve performance. The initial coding effort produced approximately 150 locks with approximately 2000 lock acquisition and deacquisition assertions.

Locking Primitives and Strategy

When we started the SVR4/MP project, we implicitly assumed that we would use traditional spin and *PSEMA()/VSEMA()* locks. There were three basic reasons for this assumption:

- 1 We could use more code from our multiprocessing UNIX System V.3 kernel in SVR4/MP if we used the locks we had used in the past.
- 2 We had used these traditional locking schemes on past multiprocessing products and felt comfortable with them.

- 3 Traditional spin and *PSEMA()/VSEMA()* locks or slight variants of these locks constitute a de facto standard in multiprocessing UNIX. Since everyone else seemed to be using them, who were we to do something different?

As the design effort continued, however, we began to become progressively more uneasy concerning these locks. Upon examination of SVR4 it was clear to us that very little if any UNIX System V.3 code could be applied to SVR4/MP. Given this, our second reason for using these locks became suspect — we soon concluded that sheer mental

```

/*
 * Acquire the specified processor lock.
 */
int
GetProcessorLock(ProcessorLock, Advice, LockInformation)
ProcessorLock_t ProcessorLock;
Advice_t        Advice;
LockInfo_t      LockInformation;

/*
 * Attempt to acquire the specified processor lock.  If the lock
 * is currently held, return immediately indicating that the lock
 * was not acquired; otherwise acquire the lock and return indicating
 * success.
 */
int
TryProcessorLock(ProcessorLock, Advice, LockInformation)
ProcessorLock_p ProcessorLock;
Advice_t        Advice;
LockInfo_p      LockInformation;

/*
 * Free the specified previously acquired processor lock.
 */
void
FreeProcessorLock(ProcessorLock, Advice, LockInformation)
ProcessorLock_p ProcessorLock;
Advice_t        Advice;
LockInfo_p      LockInformation;

/*
 * Changes the specified processor lock from a sleep to a spin.
 */
int
ChangeSleepToSpin(ProcessorLock)
ProcessorLock_p ProcessorLock;

/*
 * Change the specified processor lock from a spin to a sleep.
 */
int
ChangeSpinToSleep(ProcessorLock)
ProcessorLock_p ProcessorLock;

/*
 * Returns the processor which owns the specified processor lock.
 */
int
LockOwner(ProcessorLock)
ProcessorLock_p ProcessorLock;

```

Figure 2: APL Primitive Prototypes

laziness was not an overwhelmingly rational reason for major design decisions.

Nevertheless, we persisted that what most in the industry was doing was good enough for us. This sentiment was underscored by schedule — we simply didn't have enough time to invent a new locking paradigm. On the other hand, we also didn't have enough time for the algorithmic changes inherent with the use of the *PSEMA()/VSEMA()* nor for the changes associated with deadlock prevention for simple spinlocks within interrupt service routines. As is so often the case, it was decided that "it will take as long as it takes" and that even though we didn't have a chance, we'd try our best to make our schedules. This decision was not received warmly when presented to management.

One of the activities we had underway during the design phase of SVR4/MP was the prototype of a tightly-coupled multiprocessing kernel on a new and significantly faster processor implemented in an existing product. While debugging this prototype several errors were found that were caused by an attempt by a process to do a *VSEMA()* on a resource before another process had completed a *PSEMA()* operation on that resource. This error was fixed and the prototyping went on; however, it made us ask ourselves whether we shouldn't change that operation to a spinlock in a product since timely acquisition of that resource was important. We then asked ourselves the next question — as processors continue to get faster, wouldn't more and more spinlocks in place of sleeplocks be appropriate? Then several other questions were asked: when acquiring a resource, who should make the decision to spin or sleep? Should the decision to spin or sleep be mutable?

Processor Locks

It was eventually decided that a new set of lock primitives would have to be designed both to make our schedules and to prevent us from having to continually tune kernels as processor speeds increased. A new locking paradigm was conceived called APL's (*Advisable Processor Locks*) which attempted to take into account both our schedule requirements and our long-term kernel maintainability requirements. A *processor lock* was deemed to be simply a re-entrant lock on a per-processor basis. An APL

was deemed to be a processor lock which contained state information concerning the amount of time for which it would be locked. This state information acted as an advisory to other processes attempting to lock the same processor lock as to whether they should spin (the lock would be held for a short time) or sleep (the lock would be held for a long time).

The primitives used to manipulate APL's may be found in Figure 2.

The arguments associated with one or more of these assertions are described below:

- **ProcessorLock** — A pointer to the actual lock which is the object of the acquisition, deacquisition, modification, or identification primitive.
- **Advice** — This argument specifies whether subsequent assertions of *GetProcessorLock()* should spin or sleep. There are bits reserved in this argument to specify whether it is possible to sleep or if it is mandatory that contention on the specified lock be resolved by spinning.
- **LockInformation** — A pointer to a structure used to associate debugging and performance information with the lock specified in the lock manipulation assertion. This argument is used only during internal debugging and tuning; through conditional compilation it is not used in the production system.

The lock state of a process is maintained as part of the process's context in the user structure of that process. APL's are not held across *sleep()/wakeup()*: when a process goes to sleep holding an APL, the APL is released. When the process is awakened it must contend for the locks released when it went to sleep. The deacquisition and reacquisition of these locks are all handled within the *sleep()/wakeup()* code itself. This scheme was put in place to aid the independent parallel development of multiple subsystems — this technique reduces the coupling required when a function in one subsystem asserts a function in another subsystem.

The overhead which allows APL's to be re-entered is relatively small; however, simple spin locks (i.e., traditional spin locks which are not re-entrant) are also supported in SVR4/MP as a tuning mechanism which may be used to decrease lock

```
GetProcessorLock(&i.i_ProcessorLock, ADV_SPIN, InodeLockInformation);
i.i_flags |= I_LOCK;
FreeProcessorLock(&i.i_ProcessorLock, ADV_SPIN, InodeLockInformation);
:
:
GetProcessorLock(&i.i_ProcessorLock, ADV_SPIN, InodeLockInformation);
i.i_flags &= -I_LOCK;
FreeProcessorLock(&i.i_ProcessorLock, ADV_SPIN, InodeLockInformation);
```

Figure 3: Example of Single-Threading Uniprocessor Locks

assertion overhead. Both APL's and simple locks use a "read-read-write" locking strategy in order to decrease bus and memory utilization during spins.

General Locking Strategy

Once APL's were conceived the general locking strategy of the kernel became very straightforward. As mentioned in the introduction, coarse-grained subsystem locks and global locks for subsystems under development were deemed to obtrusive in that they significantly interfered with normal kernel operation. In addition, we had found in the past that intuitive lock placement by multiprocessing experts was an art not well understood or even agreed upon by the experts.

For these reasons we decided to use APL's to guard existing UNIX uniprocessor locks. Using this scheme an APL was used to single-thread the act of locking and unlocking a UNIX resource. An example is shown in Figure 3.

There were two basic types of existing locks to which we had to single-thread access: explicit locks as shown above in which a resource (in this case an inode) is explicitly locked for manipulation and implicit locks in which the sequential uniprocessing nature of the system was assumed to inherently protect the lock. An example of an implicit lock is a section of code in which a structure will be read only; this must be locked if the possibility exists that this structure could be modified by a process on another processor which would cause the read to be invalid.

As we expected, once the tuning of the kernel began we found many places in which the original locks led to contention. Each time one of these places was discovered it was algorithmically modified to use a scheme more suitable for efficient multiprocessing. More importantly, however, we discovered that the vast majority of the original locks did not result in contention. This was an extremely important consequence — it allowed us to concentrate our efforts on bottlenecks instead of spending a great deal of time debugging an implementation of "efficient" locking on relatively unimportant (with respect to both performance and scalability) sections of code.

The original set of locking primitives discussed in the *Processor Locks* subsection of this section have to date been sufficient for almost all of our requirements in implementing and subsequent tuning of SVR4/MP. As tuning has continued and we have continued using APL's for more efficient types of locking, we have found that a multiple reader/single writer APL is also desirable for both performance and convenience in many situations. For this reason a reader/writer APL has been implemented and used in the STREAMS subsystem.

Another performance enhancement was the inclusion of a set of atomic locking primitives which are implemented as in-line assembler macros. On machines supporting some type of lock semantics on general operations, these atomic locking primitives are used to allow single operation manipulation of simple resources. In cases in which this technique is applicable, it is much more efficient than the use of either an APL or even a simple spinlock protecting the manipulation of a resource. And unlike simple spinlocks, this technique does not suffer nested spinlock deadlock associated with interrupt service routines on a single processor.

Processor Lock Sleeping and the Thundering Herd

The algorithmic modifications to UNIX mentioned earlier due to usage of the *PSEMA()/VSEMA()* locking paradigm are a result of the fact that *VSEMA()* only wakes a single process at a time while the traditional UNIX *wakeup()* causes all processes waiting on the same resource to be awakened. In a uniprocessing system the behavior of *wakeup()* is mitigated by the fact that the first process awakened will quite often acquire and release the resource under contention before the any other awakened process is able to begin execution to contend for that resource. In multiprocessing systems, however, this behavior results in a problem known as *thundering herd* in which excessive CPU cycles are spent waking processes only to have most of those processes go back to sleep contending on the resource on which they were already asleep.

In order to obviate thundering herd behavior, a group of processes sleeping due to processor lock acquisition contention is awakened due to processor lock deacquisition only a single process at a time. This is not a problem for new code; however, the placement of processor lock primitives which can result in a sleep in original UNIX code suffers from the same requirement of algorithmic modification associated with an assertion of a *PSEMA()/VSEMA()* primitive. For this reason it was decided that placement of sleep advisories in existing code would be delayed to the tuning phase and that sleep advisories would be used only when some measurable performance degradation was noted.

Private Data

The attributes of per-processor SVR4/MP private data are given below:

- Each private data item resides at the same virtual address for each processor.
- Supported via the reserved load module private data section *pdata*.
- There is no limit on the size of the private data section.
- The private data section is replicated for each processor at kernel start-of-day.

The designation of existing data structures as private was used sparingly in SVR4/MP in order to save physical memory; however, in no case did we change the nature of an existing SVR4 data structure through the use array indices or pointers. The reasons for this are two-fold: the penalty associated with indexes or pointers and our desire not to modify existing code. Thus existing data structures such as *runrun*, *kprunrun*, *curpri*, *qrunflag*, etc. were specified as being private.

The scheme used to specify per-processor private data in SVR4/MP is a good example of a situation in which our desire to not change existing code caused conflicting requirements. In order to change as little code as possible in SVR4 a mechanism for slightly changing the declaration syntax was required, i.e., create a new declaration modifier much like the standard "static" or "unsigned" modifier. We pictured the implementation of this to be along the lines of the code segment shown in Figure 4.

```
#if defined(OriginalSVR4)
int      MyProcessorId;
int      MyProcessorIndex;
#else
private int      MyProcessorId;
private int      MyProcessorIndex;
#endif
```

Figure 4: Ideal Processor Private Data Definition

The cleanest way to implement this was to change the compiler; however, this was deemed unacceptable since we envisioned this code being used with many different compilers. Instead we decided to implement this using standard assembler macros in the manner shown in Figure 5.

In this example the variables *MyProcessorId* and *MyProcessorIndex* are specified to be private data which is four bytes in length. The macros *StartPrivateData()* and *EndPrivateData()* simply defined assembler directives specifying the beginning

and ending of the *.pdata* section while the macro *PrivateDataItem()* was an assembler directive specifying the data item as being global, aligning the data appropriately, and reserving area for the data.

```
#if defined(OriginalSVR4)
int      MyProcessorId;
int      MyProcessorIndex;
#else
StartPrivateData()
PrivateDataItem(MyProcessorId, 4)
PrivateDataItem(MyProcessorIndex, 4)
EndPrivateData()
#endif
```

Figure 5: Processor Private Data Definition With Macros

While we wanted to change the SVR4 code as little as possible, we also had a great deal of existing driver code from previous NCR multiprocessing products for which we wanted to minimize alterations. For this reason we also supported a mechanism used on earlier NCR products by which a developer could specify an entire module as containing nothing but per-processor private data. A script which detected the presence of an enabling specification in the first 100 lines of the module was brought forward into SVR4/MP. The *Makefile*'s for the kernel were then modified to execute this script which used linker directives to specify that all data in the module was private.

Interprocessor Communication

Processors communicate with each other through the use of cross processor interrupts (CPI's). There are two types of CPI's: synchronous and asynchronous CPI's. SVR4/MP provides two primitives to manipulate CPI's — the calling conventions of these primitives are given in Figure 6.

The arguments for both types of CPI assertions are identical and are described below:

- **ProcessorMask** — A bit mask specifying one or more processors, including the asserting

```
int
RequestSyncCPI(ProcessorMask, InterruptPriorityLevel, Routine, ReturnValue,
               Argument1, Argument2, Argument3, Argument4, Argument5)
ProcessorMask_t ProcessorMask;
unsigned int    InterruptPriorityLevel;
int             (*Routine)();
int             *ReturnValue;
int             Argument1, Argument2, Argument3, Argument4, Argument5;

int
RequestAsyncCPI(ProcessorMask, InterruptPriorityLevel, Routine,
               Argument1, Argument2, Argument3, Argument4, Argument5)
ProcessorMask_t ProcessorMask;
unsigned int    InterruptPriorityLevel;
int             (*Routine)();
int             Argument1, Argument2, Argument3, Argument4, Argument5;
```

Figure 6: CPI Primitive Prototypes

processor, to which the CPI is to be sent. For convenience, a per-processor private data structure called *EverybodyElsesProcessorMask* (non-inclusive of the current processor) is kept. A shared structure for *EverybodyysProcessorMask* (inclusive of the current processor) is also kept.

- **InterruptPriorityLevel** — The interrupt priority level, from 0 to 7, at which this CPI is to be delivered. An interrupt priority level of 7 is non-maskable.
- **Routine** — The function which is to be asserted on the target processor upon receipt of the CPI. If a synchronous CPI is requested then this function must complete execution on the target processor before control is returned to the routine which asserted the *Request-SyncCPI()* on the host processor; otherwise the function is executed asynchronously.
- **ReturnValue** — This value is used to return the return value of the specified function on a synchronous CPI.
- **Argument1...Argument5** — Up to five optional arguments to the routine specified by *Routine* described above are allowed.

The prototype hardware base supports only one interrupt which may be used for CPI's and it is the highest priority interrupt in the system. Software supports multiple interrupt levels by queueing CPI's which are asserted but currently masked and then by checking that queue during the assertion of *spl()* routines or at exit points from the kernel. If an appropriate CPI is found it is executed at that time. This mechanism has been found to be moderately expensive in terms of *spl()* overhead — an interrupt controller is currently in development which provides hardware support for multiple interrupt levels for CPI's.

SVR4/MP Subsystem	Lock Types	Lock Assertions
os	42	333
vfs	7	60
s5	6	34
ufs	3	20
namefs	3	7
proc	1	43
specfs	4	27
fifofs	5	24
vm	12	202
disp	5	80
STREAMS	20	124
io	13	37
TTY	9	41

Table 1 – SVR4/MP Subsystem Approximate Lock Counts

CPI's are used for start-of-day initialization, clock tick distribution, ATC coherency, the kernel debugger, and other functions. CPI's are used as little as possible in SVR4/MP due to the overhead

associated with the receipt and subsequent handling of the interrupt and its associated function.

Locking Strategy in Major Subsystems

The next few sections cover the multi-threading strategy used in several major SVR4/MP subsystems.

The approximate number of locks in SVR4/MP subsystems is given in Table 1.

The second column in the table specifies the number of lock data types while the third column specifies the number of locations in the code in which an attempt is made to acquire a lock.

General Process Management/Dispatcher

Due to the tightly-coupled, symmetric nature of the target hardware, processes are easily assigned to any processor in the system. The most simple approach to managing processes in this type of system is to share as much as possible. This is the underlying theme in the descriptions to follow in this section.

Dispatcher

The dispatcher is responsible for allocating processes to processors. Processes that are ready to run reside on the dispatcher queue. The dispatcher queue is shared between all the processors. Each processor is self-scheduling. Therefore, when a processor needs another process to run, it acquires a lock and takes the highest priority process that it can run off the dispatcher queue.

Processes may be bound to a particular processor. The processor to which a bound process is bound is the only processor that will take the process off the dispatcher queue and resume it.

An idle process is created during kernel boot for each processor in the system. If a process gives up the processor and the scheduler can find no other suitable process to resume from the dispatcher queue, the scheduler chooses the current processor's idle process. This mechanism provides each processor with a unique process state in which to idle. Idle processes never appear on the dispatcher queue.

Each process class has two locks. The first lock is acquired before traversing or manipulating the process class's *proc* list. The second lock is acquired before manipulation of the process class's parameter table.

General Process Management

Any sleeping process resides on the sleep hash queue based on the wait channel on which the process is sleeping. To allow any processor to awaken any sleeping process, this queue is shared. As with the dispatcher queue, any manipulation of the sleep hash queue requires ownership of a lock. Therefore, *sleep()*, *unsleep()* and *wakeprocs()* hold this lock

when queuing and dequeuing processes to and from the sleep hash queue.

wakeprocs() awakens all processes sleeping on the given wait channel. Since this may result in the thundering herd phenomenon, another function was developed. *WakeUpOneProcess()* scans the sleep hash queue for the highest priority process sleeping on the given wait channel. Only this process will be awakened. If no sleeping process is found, then all stopped processes will be awakened. The caller is returned information permitting it to determine whether any processes were awakened.

There is a lock associated with the process group information. All processes within the same process group are linked together and the root of each process group list can be found in the process group hash queue. Since the process group hash queue is shared by all the processors, any manipulation is performed after acquiring this lock.

All processes reside in a genealogical tree. This is implemented as parent/child/sibling links for each process. Any traversal or manipulation of this tree is preceded by acquisition of a process hierarchy lock.

The free list of credentials is shared by all the processors. Therefore, this free list as well as the reference count in each credential structure is manipulated only after acquiring a lock.

Similarly, the list of active processes (*proc* structures) and *pid* structures is not traversed or manipulated without ownership of a lock.

The *proc* structure itself contains a lock. This lock is held during any manipulation of critical fields within the *proc* structure. The list of critical *proc* structure fields includes: *p_stat*, *p_pri*, *p_flag*, *p_flag*, as well as all information pertaining to the process's signal state.

Clock

Clock interrupts are handled by one processor, the first processor to recognize the interrupt. Part of *clock*'s processing is process dependent, part is processor dependent and part is system dependent. The process and processor dependent parts of *clock*'s processing is replicated in a function that is issued to all other processors in the system via an asynchronous CPI function request. These parts include: process and processor statistics gathering, profiling, timeslicing, etc.

Virtual Memory

The virtual memory management (*vm*) subsystem was partitioned into functional layers of resources to determine an initial hierarchy for multithreading. In Figure 7, locks for the resources on the same line will never be held simultaneously. The locks for these resources must be acquired in

accordance with the ordering from top to bottom.

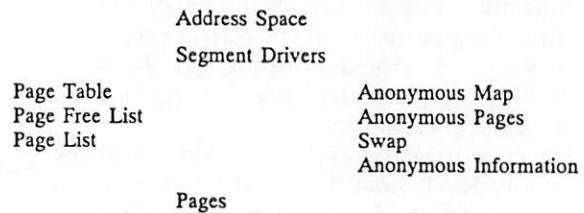


Figure 7: Initial Virtual Memory Locking Hierarchy

This partitioning helped to identify resources under the auspice of *vm* that must be protected by multiprocessor locks.

The Address Space / Segment Layer

The kernel address space (*kas*) is a shared resource; therefore, accesses to the kernel address space must be protected. The user address spaces are unique to a particular process. The user address space was not locked in the initial implementation. The processor lock for the address spaces is provided via macros (*GetAddressSpaceLock()*, *FreeAddressSpaceLock()*) and is easily adaptable to the locking of the user address space where it may be determined necessary to support sharing of address spaces such as with *threads*).

Locking of the kernel address space is necessary for the manipulation of the linked list of segments in the space. The address spaces and the attached segments are protected via the address space lock (*AddressSpace->a_Lock*).

Access to the address space structures free list (*as_freelist*) and to the segment structures free list (*seg_freelist*) must be protected. These free lists are manipulated via the *kmem_fast* functions and are therefore protected by the kernel memory locks.

The Hardware Address Translation (*hat*) Layer

The hardware address translation layer contains machine specific *hat* structures and procedures. The page table data structures (*ptdat*) contain information about page tables. The *ptdat* for a particular page table is available from the page table's associated page structure. The *ptdat* structure may be shared and must be protected. The active page table list (*ActivePageTableList*) must be protected. Mapping links (*p_mapping*) contained in the page structures are associated with page tables. These mappings link together page table entries which share a page. The manipulation of this mapping list must be protected. The page table lock *PageTableLock* controls access to these related resources.

The Segment Driver Layer

The segment drivers are segment functions which manipulate data associated with particular segment types. The segment/segment driver layer is protected in part by the address space lock. Some segment drivers reference shared system data or contain data which may be shared by other segments. The shared data must be protected.

The user structure segment driver must protect access to the free list of user structures in the system user structure array. The user structure segment lock *SeguLock* controls access to the system user structure array.

The *vnode* segment driver must protect access to the potentially shared anonymous mapping associated with that segment. The block of memory pointed to by the anonymous pointer in the anonymous map structure may require a new block of memory for growth during creation of a segment (*segvn_create*) resulting in the anonymous pointer changing to point to the new block. Therefore, access to the anonymous mapping array by portions of code, such as shown in Figure 8, must be protected.

The anonymous map lock named *AnonymousMap->Lock* protects the segment anonymous mapping of anonymous pages.

The *vnode* segment driver must also protect accesses to the anonymous map structure free list (*anonmap_freelist*) and to the *vnode* segment structure free list (*segvn_freelist*). These free lists are manipulated by the *kmem_fast* functions and are therefore protected by the kernel memory locks.

The Physical Page Layer

The physical page layer includes several critical resources. The page lists — page free list, page cache list, page active/hash/buffer list — must be protected during manipulation. Tracking variables — *availrmem*, *availsmem*, *pages_pp_kernel*, *pages_pp_locked*, *freemem* — must also be protected. The access to the page free list and the page cache list is controlled by the page free list lock *PageFreeListLock*. This lock also controls the tracking of *freemem*. The access to active page lists (active/hash/buffer pages) is controlled by the page list lock *PageListLock*. The tracking variables (with the exception of *freemem*) are protected by the lock for page status tracking *PagesLock*.

The Anonymous/Swap Layer

The anonymous pages are physical pages which have no relation to a file system. These pages are backed exclusively by swap space as opposed to having an association with a file system. The manipulation of the content of the page structures must be protected (reference counts, data pointers, etc.). The manipulation of the anonymous pages is controlled by the anonymous page lock *AnonymousPageLock*. The tracking information (*anoninfo* structure) must also be protected. The tracking for the anonymous pages is controlled by the anonymous information lock *AnonymousInformationLock*.

The swap space is closely related with the anonymous pages. The swap space is a linking of swap areas (*swapinfo* structure) which contain arrays of anonymous pages. The access to these swap areas must be protected. The swap lock *SwapLock* controls the access to the concatenated (linked) swap areas.

Statistics in Private Data

For accurate statistics, certain statistical flags and data must be protected. The virtual memory statistics (*os/vm_meter.c*, *sys/vmmeter.h*) are located in processor private memory and do not require locks.

Miscellaneous

The buffer list for aborting pages associated with failed requests (*abort_buf*) must be protected. The abort buffer lock *AbortBufferLock* protects the aborted buffer list.

The *cleanup* buffers (*bclnlist*) must be protected during manipulation via the *cleanup()* routine (initiated from *pageout()*, *sched()*, *page_cv_wait()*, *page_lookup()*, *page_get()*). Checks for calling *cleanup()* and *cleanup()* itself control access to the *bclnlist* by the *cleanup* buffer list lock *CleanupBufferListLock*.

Special Memory Management Considerations

The swapping of an address space associated with a process requires special locking considerations. The stealing of pages by the *pageout* daemon also requires considerations to prevent the stealing of pages potentially useful to active address spaces. Both of these memory management procedures leads to concerns for translation cache (ATC/TLB)

```
AnonymousPagePointer = &SegVN_Data->AnonymousMapping->anon[SegVN_Data->anon_index];
:
:
while (*AnonymousPagePointer++)
    doSomething();
```

Figure 8: Unsafe Traversal of Anonymous Map Structures

coherency. These concerns were not addressed in full detail in the initial implementation.

Locking Hierarchy

The original partitioning did not properly address the manipulation of the kernel address space. Segment drivers such as the *vnode* segment driver would enter the kernel address space via other segment drivers such as the *kmem* segment driver. This problem also led to the realization that the user address space is unique; therefore, this address space did not require locking. Other hierarchy problems were revealed due to interaction with physical pages and anonymous pages. These problems were revealed through the use of the *lockinfo* debug structure and the lock debug driver. The current locking hierarchy is illustrated in Figure 9.

STREAMS

The strategy for making the STREAMS subsystem multithreaded involves utilizing locks to protect critical data structures such as the queue structure. Locks must also be used to protect link lists and all the links in the chain. For example, the *q_link* and *q_next* fields of the queue structure are used for linking queue structures together. Therefore, there is one lock for all the non-linking fields in the queue structure and other locks to link queue structures together.

The critical resources used by the STREAM subsystem must be protected by multiprocessor locks. Any modification of these resources must take place while the appropriate lock is owned by the processor performing the modification. Any access to these resources that requires their contents to remain unchanged for a period of time must take place while the appropriate lock is owned by the processor performing the access.

Resource Allocation Lists

Nearly all the memory resources required by STREAMS are dynamically allocated using *kmem_alloc()*. These dynamically allocated data structures are then maintained in a linked list. When

they are no longer needed, they are removed from the list and freed using *kmem_free()*.

Listed below are the allocation list locks and the scope of their locks.

- The allocated list of headers for a stream(struct stdata) is protected by the lock *STREAM_HeadFreeListAccessLock*. The statistics in *strst.stream* are updated under this lock also.
- The allocated list of queue structures is protected by the lock *STREAM_QueueFreeListAccessLock*. The statistics in *strst.queue* are updated under this lock also.
- The allocated list of message block descriptors is protected by the lock *msgfreelist_AccessLock*. The statistics in *strst.msgblock* are updated and the debug functions *insert_msg_inuse()* and *delete_msg_inuse()* executed under this lock also.
- The allocated list of data block descriptors is protected by the lock *mdbfreelist_AccessLock*. The statistics in *strst.mdbblock* are updated and the debug functions *insert_mdb_inuse()* and *delete_mdb_inuse()* executed under this lock also.
- The allocated list of link blocks is protected by the lock *STREAM_LinkBlockAccessLock*. The statistics in *strst.linkblk* are updated and the mux_node list is maintained under this lock also.
- The allocated list of stream event descriptors is protected by the lock *STREAM_EventCellFreeListAccessLock*. The statistics in *strst.strevent* are updated and the stream event cache is maintained under this lock also.
- The allocated list of queue priority band structures is protected by the lock *STREAM_QueueBandFreeListAccessLock*. No statistics are maintained for qband structures.

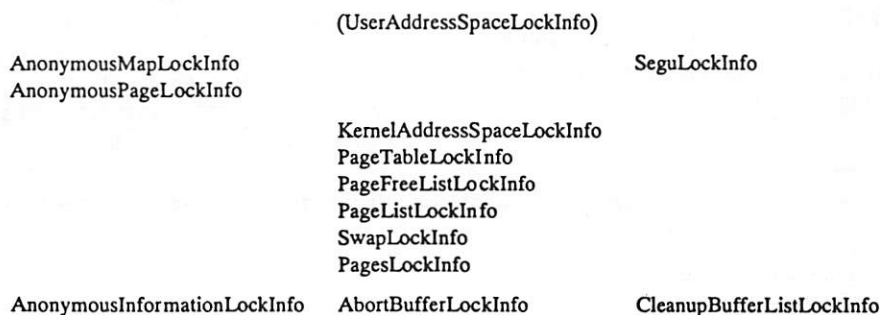


Figure 9: Current Virtual Memory Locking Hierarchy

The Queue Structure

The *queue_t* pairs are assigned to each instance of a module or driver that is being pushed or opened. The *queue_t* pair contains message queues, state information, and links to other queue structures for each of the upstream and downstream *queue_t*'s for the module.

The queue structure is protected by the lock *q_AccessLock*, an element of the queue structure. Reading a field of the queue structure does not require getting the lock unless an action depends on the value not changing. The message queue and the priority band queue are also covered by this lock. No message block or data block descriptor has a lock because they are either linked to a queue or are being passed to neighboring module. The *q_link* field is not covered by this lock but is covered by the STREAMS scheduling lock described below. The *q_next* field is not covered by this lock but is covered by the STREAMS configuration lock described below.

The Stream Head Structure

The stream header contains all the information required to interface the stream to the rest of the operating system. The stream header structure is protected by the lock *sd_AccessLock*, an element of the stream header structure. Reading a field of the stream header structure does not require getting the lock unless an action depends on the value not changing.

In order to reduce the impact of multithreading the stream head code, the *sd_SysCallSerializationLock* is used so only one system call per stream head can be active at a time. With this lock, the *sd_AccessLock* need only be obtained where *splstr()* protection is also needed.

Queue Scheduling

The data structures for queue scheduling are *grunflag*, *queueflag*, *qhead*, *qtail*, *qbf*, *scanqhead*, *scanqtail*, and *strscanflag*. These are all flags or head nodes of queues. These data structures reside in private data, so no locks are required for any of these data structures.

The bufcall List

The data structures for *bufcall()* handling are the flags *strbcwait* and *strbcflag*, and the head nodes for the lists of *strevent* structures, *strbcalls*. The data structures for *bufcall()* handling are protected by *bufcall_AccessLock*.

Other Resources

The pending *ioctl* messages and each multiplexor link is assigned a unique identifier. The next available identifier for each is maintained in *ioc_id*

and *lnk_id* respectively. These identifiers are covered by a simple lock. The data field *strcount* is the byte count of all the dynamically allocated STREAMS resources. This field is also covered by a simple lock. The *strst* structure contains all the relevant stream statistics. The maintenance of this structure is covered by the lock relevant for the statistic being updated.

STREAMS Plumbing

Changing the configuration of a STREAMS stack while messages are flowing through the stack has proven to be the most difficult problem to solve in multithreading STREAMS. Many of the solutions for localizing a lock either have been ineffective or have required many additional changes to existing code in order to become multithreaded.

Messages may be moving through the stream while the *qattach()* and *qdetach()* operations are taking place. This results in *q_next* pointers becoming NULL unexpectedly and the *q_ptr* being invalid causing the module *put* and *service* procedures to panic. In the *qattach()* operation, for example, the *put* procedure can be entered before its open function has had a chance to execute. In a multiprocessing system the following sequence of operations for *qattach()* must be protected from other threads of execution in order to avoid an uninitialized *q_ptr* field in the queue structure:

- linking of the new module below the stream head;
- initialization of queues;
- call to open function.

In the *qdetach()* operation the reverse is true. The *put* and *service* procedure can be entered after its close function has executed. In a multiprocessing system the following sequence of operations for *qdetach()* must be protected from other threads of execution in order to avoid an invalid *q_ptr* field in the queue structure.

- running of *service* procedures, if necessary;
- call to close function;
- removal of any scheduled *service* procedures for the module;
- unlinking of the module from the stream head.

The message flow through the stream must be stopped during the *qattach()* and *qdetach()* operations for the stack being reconfigured. Exercising flow to stop messages is not feasible because high priority messages are not stopped by current flow control mechanisms. Stopping messages in a given stack is difficult and maybe insufficient when single threaded modules are supported so a single lock is used for all STREAMS reconfiguration.

The *STREAM_PlumbingLock* is used to stop all message flow through streams until the reconfiguration has been completed. This lock protects the *q_next* field of every queue structure active

in the system. Unlike the other locks in streams, this lock can be held by multiple processors in a read-only mode but only one processor can hold the lock as a writer.

This lock is obtained in read-only mode for STREAMS interrupt service routines, *service* procedures, system calls, and *timeout* and *bufcall* functions.

The *qattach()* and *qdetach()* procedures obtain the plumbing lock in write mode to perform the reconfiguration operations.

Multithreaded Modules and Drivers Requirements

Implicit STREAMS Assumptions

Conventional STREAMS modules and drivers contain many dependencies which are based on the assumption that they are running on a single processor. Below is a summary of the significant assumptions which are made by conventional STREAMS modules and drivers concerning preemption and reentrancy.

- Any routine module or driver can avoid preemption by simply raising the interrupt priority level with *splstr()*.
- A module or driver running one of its *put* procedures can be reentered by its *timeout()* routines. This can be avoided by simply raising the interrupt priority level.
- A module or driver running one of its *service* procedures can be reentered by its *put* routines. This can be avoided by simply raising the interrupt priority level.
- A module or driver should assume that it will be reentered while calling an adjacent module's *put* procedure due to in-line processing of *putnext()*.
- Module and driver *put* procedures can always assume that they will not be preempted by any *service* procedure regardless of interrupt priority level.
- Module and driver *open* and *close* routines

and *service* procedures can be assumed to be non-reentrant.

Multithreading Implications

Most of the basis for the above assumptions break down in a multiprocessing environment. The following list shows the implications of a multithreaded STREAMS subsystem.

- A module or driver's *put* and *service* procedures can be executing simultaneously, even for the same queue.
- It can be assumed that access to STREAMS queues shall be synchronized when manipulated by STREAMS specific utility routines (e.g., *putq()*, *getq()* etc.). In other words, All STREAMS related utility routines that manipulate or otherwise examine queues shall lock them before doing so and unlock them when finished.
- Raising the interrupt priority level with *splstr()* is still required to avoid preemption on a given processor because locks can be secured by the interrupt thread of execution.

Synchronization Guidelines

The name of the STREAMS queue resident processor lock is *q_AccessLock*. This lock should not be needed by the driver developer very often since STREAMS utility routines exist for performing nearly any kind of STREAMS queue manipulation. The following rules apply for using these locking mechanisms in a STREAMS module.

- Interrupt priority level must be held at *splstr()* throughout the time that a *q_AccessLock* is held.
- A *q_AccessLock* must not be held while sending a message to an adjacent module. If this were allowed it would most certainly cause deadlocks.
- Two *q_AccessLocks* may not be held at the same time except in the case where a driver or module needs to lock its read queue and its write queue simultaneously. If this must be done, the read queue must always be locked

```
while ( ILOCKED bit is set in the inode's i_flag member ) {
    set IWANT bit in i_flag
    sleep on address of inode
}
set ILOCKED bit in the inode's i_flag member
```

Figure 10: Unsafe Uniprocessor Sleep Lock

```
GetProcessorLock(in the vnode enclosed by the given inode)
while ( ILOCKED bit is set in the inode's i_flag member ) {
    set the i_want member of the inode to 1
    sleep on address of inode
}
set ILOCKED bit in the inode's i_flag member
FreeProcessorLock(in the vnode enclosed by the given inode)
```

Figure 11: Parallelized Uniprocessor Sleep Lock

first.

- It is required that data structures local to modules or drivers not remain locked across calls to adjacent modules.
- STREAMS modules and drivers may sleep only in their *open* and *close* procedures.
- Message blocks need not be locked since they may be held by only one queue at a time.
- Data blocks referenced by more than one message block should have a synchronization mechanism of their own.

VFS and File Systems

The Virtual File System (VFS) and the implementation-specific file systems³ are multithreaded to allow concurrent execution on an arbitrary number of processors except in critical regions of code, where a data structure shared by multiple processors is modified or where the value of the data

³The implementation-specific file systems multithreaded for the system discussed in this paper were *s5*, *specfs*, *fifofs*, *namefs*, and *procfs*.

structure must not change for some interval. Most of these critical regions are serialized on a per-data-structure basis. For example, only one processor at a time can use *VN_HOLD()* to change the reference count in a particular vnode, but one processor can execute *VN_HOLD()* to change the count in vnode *vn1* at the same time another processor runs *VN_HOLD()* on vnode *vn2*.

With the exception of mounts and unmounts, multiprocessing in the file systems adheres to the principles of VFS architecture; in particular, implementation-specific file system code is allowed to decide what system-level locking is necessary.⁴ That is, the VFS layer makes no effort to serialize calls to the VFS operations associated with a particular implementation-specific file system or to the vnode operations associated with its vnodes. Moreover, the semantics of VFS and vnode operations are unchanged with respect to the AT&T uniprocessing code, even where changes would more fully take advantage of the multiprocessing feature. For

⁴*File System Type Writer's Guide*, page 2.

Processor Type:	MC88100	Hardware Cache Coherency:	Yes
Processor Speed:	25MHz	Hardware ATC Coherency:	No
Number Of Processors:	1, 2	I/O Bus:	Multibus-II
Cache Device Type:	MC88200	Interrupt Symmetry:	Yes
Per-Processor Cache Size:	32K	I/O Symmetry:	Yes

Figure 12: Performance Analysis System Configuration

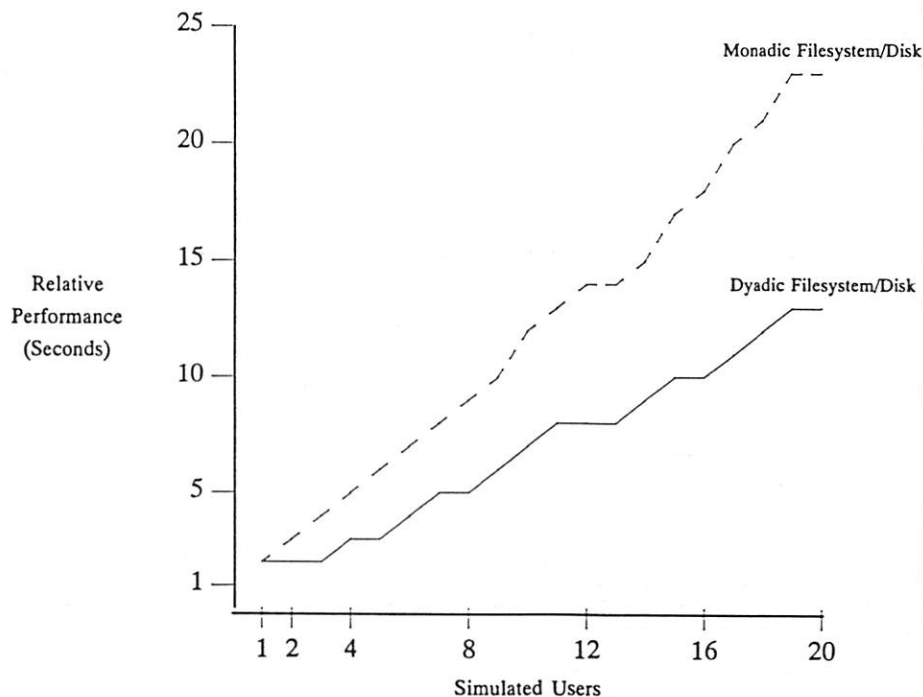


Figure 13: Monadic and Dyadic Filesystem/Disk Scalability

example, in both the uniprocessing kernel and in the multiprocessing version, reading an *s5* file is atomic in the sense that *IRWLOCK()* is asserted on its inode. Even if the read sleeps, no other process or processor may read the file until the first read is finished. But with the multiprocessing feature, distinct files may be read concurrently.

Two methods are used to synchronize the critical regions of code. In cases where process synchronization is not an issue in the uniprocessing source, we use the multiprocessor synchronization techniques directly. For example, before we clear the *v_stream* member in the *vnode* of a streams device we're closing, we acquire the processor lock in the *vnode*.

In cases where process synchronization is an issue in the uniprocessing source, the techniques used for one process to protect a data structure from other processes that may get switched in are bolstered to provide protection from processes running concurrently on other processors. An example is *ILOCK()* in the *s5* file system code. The familiar paradigm shown in Figure 10 has been changed to the paradigm shown in Figure 11.

The first change to the paradigm introduces the use of a processor lock in the structure we desire to control exclusively. Holding this lock allows us to check the state of the structure and, if the structure is not controlled by another process, change the state to show ownership.

The second change concerns the way a process shows its desire to control a structure that is already owned by another process. In the new paradigm, rather than setting the *IWANT* bit in the *i_flag* member, we set to 1 a member added to the structure specifically to hold the wanted status of the structure. In this way, once a process sets the *ILOCKED* bit in the *i_flag* member, it can modify the flags to its heart's content without holding the processor lock for the structure. Without the separate *i_want* member, this would be impossible; the owning process might change the flags only to have them overwritten with outmoded flags of a contending process that only wants to set the *IWANT* bit.

The only alternative to a separate wanted member is always to acquire the structure's processor lock before accessing the flags member. As the flags member is changed frequently, both in terms of instances in the source and in terms of run-time changes, this approach is bad for source maintainability and for run-time efficiency. About the only advantage it offers is that it does not require a new member to be introduced into the structure in question. Conceivably, this factor could be important, particularly in the arena of conformance to binary level compatibility standards. However, to date, we have found no cases where we needed a wanted member and could not append it to a structure, even in the case of structures specified by standards, e.g. DDI/DKI specification of the *buf* structure.

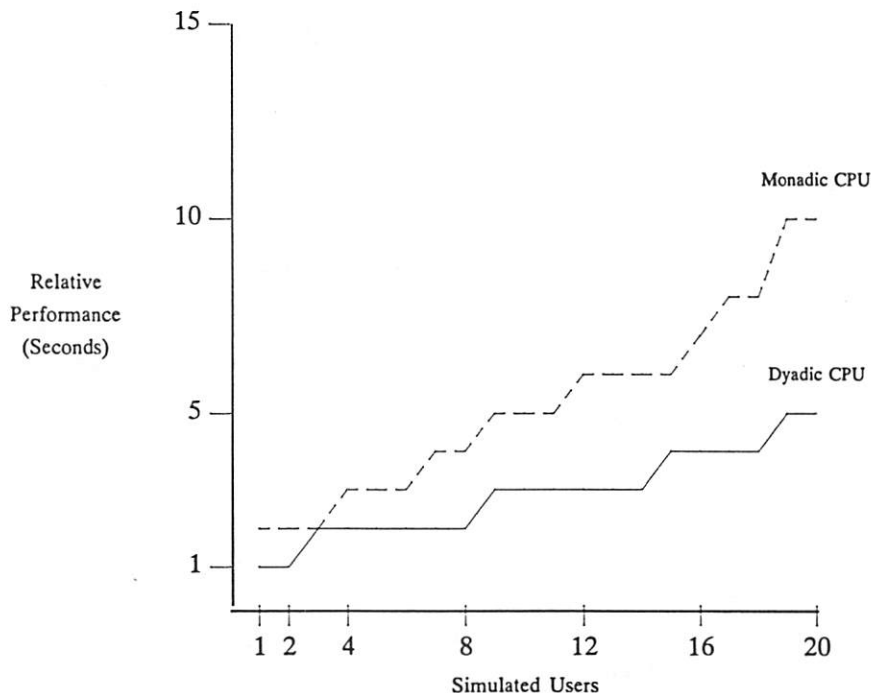


Figure 14: Monadic and Dyadic CPU Scalability

Performance Analysis

Three benchmarks are presented to characterize the performance of SVR4/MP: an NCR proprietary benchmark designed to simulate a multi-user commercial environment for both CPU and filesystem/disk intensive applications and which simulates from 1 to 20 users, the NCR System Characterization Benchmark (SCB), and the TP1 debit/credit benchmark executed on the Oracle relational database management system. These benchmarks were chosen to represent widely varying system workload profiles.

The system configuration is described in Figure 12.

All benchmark results are specified in terms of scalability and are normalized to the single processor result of the respective benchmark.

The scalability of the commercial filesystem/disk benchmark is shown in Figure 13 while the scalability of the commercial CPU benchmark is shown in Figure 14. Both support from 1 to 20 simulated users. The filesystem/disk benchmark primarily performs various types of I/O operations (e.g., *read(2)*, *write(2)*, *sync(2)*, *fsync(2)*, *mmap(2)*, etc.) operations; the CPU benchmark primarily performs user and non-I/O system call activity.

Both the CPU and filesystem/disk benchmarks display nearly linear scalability as the number of users increase. The filesystem/disk benchmark has an associated scalability factor of 1.8 at 20 simulated users. Given that well over 90% of the time associated with the benchmark is spent in the operating system, we were quite pleased with this result. Further study, however, showed that 15% of the 20% degradation for the second processor was due to a hardware constraint in the prototype system — with this constraint lifted the actual scalability is estimated to be 1.9-1.95. The CPU benchmark has an associated scalability of 2.0 at 20 simulated users.

The SCB benchmark showed scalability of 1.94. This is especially impressive given the large number of system calls and I/O involved in SCB.

In past efforts we have found scaling the TP1 benchmark to be one of the greatest challenges for a multiprocessing system. The final TP1 benchmark results showed a scalability of 1.72 for 4 generators, 1.72 for 8 generators, and greater than 2 for 12 or more generators. TP1 is a non-trivial benchmark; tuning both the system and the database results in an optimal configuration for a given workload. This is the reason that superlinear scalability is exhibited for 12 or more generators.

Conclusion and Future Work

SVR4/MP was completed within a month of its original schedule and met all of its scalability goals. The primary reasons for this success were a series of

goals and corresponding technological innovations which made the achievement of those goals possible. These goals and their corresponding driving technology are given below:

- Minimal code modifications. This was possible due to the creation of advisable processor locks.
- Quantitative lock placement. This was possible due to a set of hierarchical lock debug and performance tools which were taken from previous NCR multiprocessing efforts and enhanced.

In addition, SVR4/MP simply would not have been achievable without a tremendous work ethic on the part of the developers participating in its creation. In particular, the 14 on-site developers who worked incredibly long hours and participated in the construction of the core of the kernel were the primary reason for the timeliness of SVR4/MP. Without the teamwork, discipline, and adaptability of these developers the goals of minimal code modification and quantitative lock placement could not have been attained. A pivotal role was also played by management: they committed the staffing necessary for this task, provided the resources necessary for a large number of developers to work on SVR4/MP, and brought third-party application vendors into NCR to allow tuning of SVR4/MP and early availability of these applications.

There are a number of features that SVR4/MP lacks which are either currently under development or are planned for some time in the future. The most obvious area for future work in SVR4/MP is in increasing its scalability for ever greater numbers of processors. New types of locks, lock granularity modifications, algorithmic modifications, inter-processor communication enhancements, advanced ATC management, and affinity scheduling techniques are all currently under development. Portability is being evaluated through the port of SVR4/MP to different platforms and processors. Features such as loadable device drivers, C1/B1-level security, threads, and POSIX 1003.4 real-time support are all currently in the evaluation/prototype stage.

The target machine for which SVR4/MP was initially developed was an MC88000-based machine which was never released by NCR; however, SVR4/MP was subsequently ported by NCR to a multiprocessing Intel i80x86-based platform. This was then delivered to the UNIX International Multiprocessing Workgroup for further development and is slated for general distribution by UNIX System Laboratories in early 1991.

Acknowledgements

As mentioned several times in this paper, developing SVR4/MP would not have been possible without the incredibly strong work ethic of those involved. In addition to the authors of this paper, Roger Collins, Jon Cassorla, Steve Hurlbut, and Niels Brandt all participated in the on-site development team and worked directly on various kernel subsystems. Myron Merritt was the manager of this project and provided a great deal of insight into the rationale of past NCR kernel work as well as guidance in the creation of SVR4/MP. All of these people were indispensable in our realizing SVR4/MP as efficiently and quickly as we did.

Finally, we'd like to thank Jerry Butler for providing the staffing that allowed us to succeed and Mike Gerhold for the continuing exhortations to push just a little bit harder.

References

- 1 Bach, M., Buroff, S., "Multiprocessor UNIX Operating Systems", *AT&T Bell Laboratories Technical Journal*, October 1984.
- 2 Boykin, Joseph, Langerman, Alan., "The Parallelization of Mach/4.3BSD: Design Philosophy and Performance Analysis", *Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, October 1989.
- 3 Hamilton, G., Code, D., "An Experimental Symmetric Multiprocessor ULTRIX Kernel", *Proceedings of the Winter USENIX Conference*, 1988.
- 4 Sinkewicz, U., "A Strategy for SMP ULTRIX", *Summer 1988 USENIX Conference Proceedings*, 1988.

Mark Campbell was the front-line manager of SVR4/MP and is currently the Director of Systems Architecture at NCR Corporation's E&M Columbia facility. His interests include all aspects of hardware and software architecture and design. Mark received his B.S. and M.S. degree in electrical engineering from the University of South Carolina in 1983 and 1984, respectively. Mark's electronic mail address is mark.campbell@ncrcae.Columbia.NCR.COM and his U.S. mail address is NCR Corporation; 3325 Platt Springs Road; West Columbia; South Carolina, 29169.

Richard R. Barton is a Consulting Analyst for NCR Corporation. His interests include parallel processing software and architectures. Prior to his employment with NCR, Richard was a Senior Software Engineer in the Operating Systems Group at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign. He also worked for IBM-FSD, Houston, on the Telemetry Planning and Evaluation

subsystem of the USAF Data Systems Modernization project. Richard was involved in research on automatic file migration at the University of Illinois at Urbana-Champaign where he received his M.S. degree in Computer Science in 1982. In 1980, he graduated from the University of South Carolina where he received his B.S. in Computer Science. He is a member of the IEEE, IEEE Computer Society, ACM, and AAAS. Richard may be reached at Richard.Barton@Columbia.NCR.COM or via U.S. Mail at Richard R. Barton; 3325 Platt Springs Road; West Columbia, SC 29169.

Jim Browning is a Technology Consultant with the Systems Architecture organization at NCR's Engineering and Manufacturing division in Columbia, SC. He has been involved with operating systems research and development within NCR for the past twelve years and currently represents NCR on the UNIX International Technical Steering Committee. He received a B.S. degree in Computer Science and Mathematics from Mississippi State University in 1978.

Dennis Cervenka is a Senior Principle Programmer Analyst at NCR's Tower Systems Development Plant in West Columbia SC. Dennis's interest include system security, interrupt and exception handling design and development, multiprocessor startup and project management. He is currently involved in the opens systems development of a system security for UNIX V.4. In the past, he has developed/worked on disk drivers, memory management, interrupt and exception handling, multiprocessing start of day code. Dennis received a B.S. degree in Computer Science from University of South Carolina in 1985. Reach Dennis at dennis@ncrcae.Columbia.NCR.COM, or via U. S. Mail at Systems Software Group; 3325 Platt Springs Road; West Columbia, SC 29169.

Ben Curry is a Senior Principle Programmer Analyst for NCR in Columbia, South Carolina. He is currently working with the Intel Consortium in Santa Clara, California, multithreading the the file system portion of UNIX System V Release 4. He received a B.A. degree in Russian Language and Literature from Dartmouth College in 1981 and an M.S. degree in Computer Science from the University of North Carolina at Chapel Hill in 1984. Reach him electronically at Ben.Curry@ncrcae.ncr.com.

Todd Davis is a Consulting Analyst for NCR Corporation. His interests include operating systems, LAN software, and device driver interfaces. He is currently leading the development of testware for multiprocessor hardware. In the past, he developed an executive for intelligent I/O adapters and has multithreaded the file system and STREAMS for UNIX System V Release 3. He received a B.E. degree in Industrial Engineering from the Georgia

Institute of Technology in 1973 and a M.S. in Computer Science from University of South Carolina in 1981. Reach Todd Davis at Todd.Davis@Columbia.NCR.COM, or via U. S. Mail at NCR Corporation - Engineering and Manufacturing - Columbia; 3325 Platt Springs Road; West Columbia, South Carolina 29169; Office: (803)791-6863; Lab: (803) 791-6550.

Tracy Edmonds is a Computer Engineer in the Systems Architecture group at NCR E&M Columbia, SC. His interests include multiprocessor operating system internals, system level simulation and VLSI design. In the past, he developed several STREAMS-based TTY drivers for the NCR TOWER line of supermicrocomputers as well as the multi-threading of NCR's version of UNIX System V Release 4 STREAMS subsystem. He is currently doing behavioral level modeling of some custom chips for NCR's next generation systems. Tracy Edmonds received his B.S. degree in Electrical/Computer Engineering from the University of South Carolina in 1988. Reach Tracy electronically at t.edmonds@Columbia.NCR.COM.

Russ Holt holds a Bachelor of Information and Computer Science degree from the Georgia Institute of Technology. He began with NCR in September of 1983 in operating system utility development for the TOWER 1632. Russ has performed maintenance and development on TOWER Local Area Networking products (TOWERNET, DRS) and TOWER kernel software. He was a project leader for the development of the NCR SVR4/MP UNIX. He is currently a manager of system software development. (russ.holt@Columbia.NCR.COM)

John R. Slice is a Consulting Analyst with NCR E&M Columbia, SC. His research interests include parallel computer architectures, operating systems design, computer aided software development, programming languages and compilers. Currently he is the senior consultant for NCR working with the Silicon Valley based Intel Consortium, a group of companies involved in a cooperative effort with AT&T to produce a multiprocessor version of System V Release 4.0 UNIX. In the past, he was instrumental in the development of the NCR Tower 800-850 line of multiprocessor based general purpose computers for which he received an NCR Corporate Award for Meritorious Achievement. He received the B.S. and M.S. degrees in electrical and computer engineering from the University of South Carolina in 1983 and 1985, respectively. He can be reached electronically at John.Slice@ncrcac.ncr.com and by U.S. Mail at NCR Corp. Dept. 790; 3325 Platt Springs Rd.; West Columbia, SC 29169.

Tucker Smith is a Consulting Analyst at NCR's Tower Systems Development Plant in West Columbia SC. Tucker's interest include fault tolerant

system's, file system design and development, random/sequential access drivers and multiprocessor kernel debuggers. He is currently involved in the opens systems development of a fault tolerant file system for UNIX V.4. In the past, he has developed real-time transaction processing utilities to aid UNIX V.3.2 kernel performance in real-time environments. Tucker received a B.S. degree in Computer Science from Mississippi State University in 1978. Reach Tucker electronically at tucker@ncrcac.Columbia.NCR.COM. Reach him via U. S. Mail at Systems Software Group; 3325 Platt Springs Road; West Columbia, SC 29169.

Rich Wescott is a Consulting Analyst for NCR Corporation at E. & M. Columbia in South Carolina. His primary focus is the UNIX kernel and is working on the development of a product to use the Multiprocessing UNIX kernel. He received a B.S. degree in Mathematics from Drexel University in 1976. Reach Rich at rich.wescott@NCR.COLUMBIA.COM. U.S. Mail at E&M Columbia; 3325 Platt Springs Rd.; West Columbia S. C., 29169.

A NonStop UNIX Operating System

Peter Norwood – Tivoli Systems, Inc.

ABSTRACT

Tandem's Integrity S2 is a fault-tolerant computer system that provides the benefits of a standard UNIX operating system. The fault-tolerant capabilities of Integrity S2 are realized through a combination of hardware and software. The hardware supports fault-tolerant operation through a variety of techniques including triple modular redundancy, duplexed hardware, and self-checking circuitry. The operating system, NonStop-UX, is based on an AT&T UNIX V.3 kernel. It has been enhanced in a number of ways to improve the robustness of the UNIX operating system and to support fault-tolerant system operation.

UNIX and Fault Tolerance

The Integrity S2 represents a new class of computers that are attempting to satisfy the needs of users requiring both fault-tolerance and an industry standard operating environment. Fault-tolerant systems with proprietary operating systems have been successfully sold to users requiring high availability and data integrity for some time. The challenge has been to provide a UNIX solution for the fault-tolerant marketplace. Many different approaches have been tried. These attempts have ranged from systems with high degrees of fault tolerance with operating systems only superficially resembling UNIX to systems with only slight modifications to UNIX but exhibiting few of the features required for fault tolerance.

Integrity S2 successfully combines the features of fault tolerance with a fully conforming UNIX operating environment. It departs from the traditional architectures employed to provide fault-tolerance. Those traditional architectures use a proprietary distributed operating system.

The Traditional Approach

The proprietary Tandem NonStop systems [1] are a good example of the traditional approach to providing a fault-tolerant computing system. The architecture of the NonStop systems consists of a loosely coupled hardware architecture and a distributed operating system.

The hardware architecture consists of a network of computing elements, where a computing element is defined as a processor, a memory, and an I/O channel connected to I/O controllers for peripheral devices. The processors are connected via a high speed message-passing bus. All of the computing elements are designed with self-checking logic design so that hardware faults can be identified and isolated to a particular processing element.

Another processor always has access to disk resident data from a failed processor via dual ported disks and controllers. The Guardian operating system supports fault tolerance by ensuring the

properties implied by a transaction based file system - atomicity, durability, and consistency [2].

The Guardian operating system also provides support for software state checkpointing [3]. Checkpointing is the transferring of state from one processing element to another so that a computation can be restarted from the beginning of the last checkpoint on another processor.

Problems with the Traditional Approach

While the traditional approach does provide high availability and high data integrity, the two basic features of a fault-tolerant system, it poses certain problems for a UNIX implementation. Software checkpointing requires one of two things. It requires either changes to the user program to explicitly checkpoint the state of the program or it requires an operating system capable of communicating sufficient program state that user programmable checkpointing would not be required.

A fault-tolerant system that required user level changes to programs for checkpointing of software state would not be able to leverage one of the key advantages of standardization, namely the availability of off-the-shelf software packages. If we were to provide a fault-tolerant system that ran UNIX and still required applications to be responsible for checkpointing their state, then it would be impossible for customers to run off-the-shelf third party software packages and expect an increase in high availability or data integrity. Since leveraging third party software efforts is a key to success in the UNIX marketplace, this approach is not terribly attractive.

Transparent checkpointing of a user level program's state is intriguing and possible [4]. The possibility of this kind of solution is greatly increased by the evolution of UNIX into a multi-process, message passing operating system (similar to Guardian). The evolution of UNIX into a Mach [5] or Chorus [6] variant encourages the speculation that this kind of capability is possible within a UNIX framework.

Nevertheless, the current design of the UNIX operating system and the current pressures to leverage third party software packages require a radically different approach from the traditional one. In fact, our development team assumed that neither user level program state checkpointing nor transparent operating system checkpointing were a possibility. Therefore, we followed a methodology that involved improving the robustness of the UNIX operating system. This methodology will be described below.

The Integrity S2 Architecture

The goals of the Integrity S2 were to design a system that ran an industry standard version of UNIX that was capable of withstanding any single point of failure in the system. This implied that the system should have continuous availability in the event of a failure and that the data integrity of the system would not be compromised by any failure. Continuous system availability in the event of any arbitrary failure also implied a system that was

serviceable on-line, i.e. any component that failed could be repaired while the system was running.

The Integrity S2 is able to survive hardware failures because of a hardware architecture that employs triplicated processors and a duplexed I/O subsystem. The triplicated processors identify and isolate errors by voting their outputs while the duplexed I/O subsystem identifies and isolates errors through the use of self-checking circuitry.

The hardware architecture assumes a correct design and protects against failures to components in the design with different forms of redundancy. Once a faulty component is identified it can be replaced while applications continue to execute on the system. The software architecture is somewhat different.

There are three identical instruction streams executing on the system during normal operation. A coding error in the kernel could cause all three instruction streams to fail in an identical fashion thus compromising the availability of the system. We

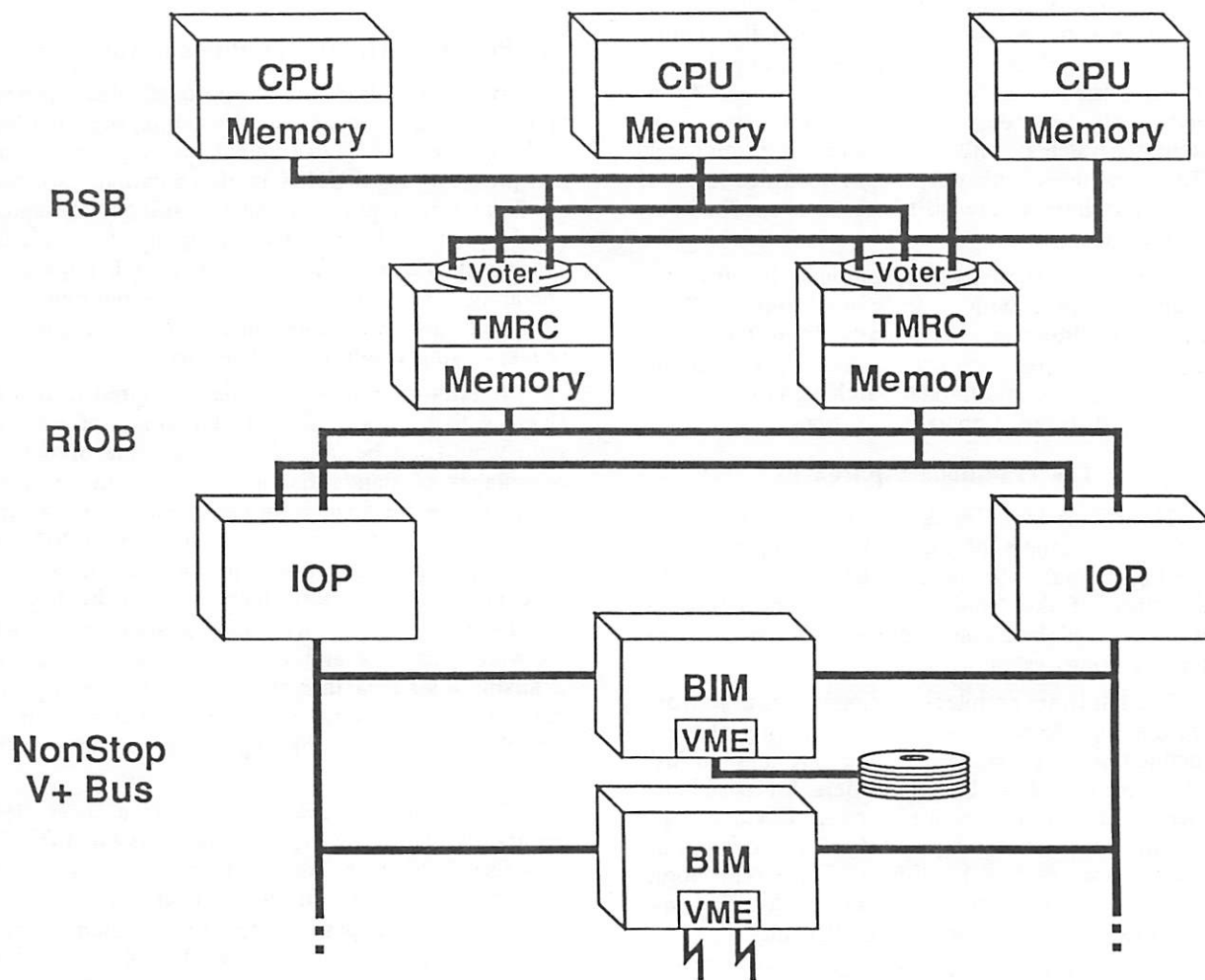


Figure 1: Integrity S2 Architecture

assumed that it was unrealistic to provide a provably correct kernel on the system. Therefore, we created a software architecture that focused on enhancing the kernel with the ability to recover from errors in logic.

The software architecture uses various techniques to identify software faults, isolate them and then attempt a forward recovery action. No software state checkpointing is used so it is not possible to rollback to a prior consistent state and continue execution from that point. However, it is often possible to identify a process that is responsible for the failure and force it to terminate or to reinitialize a portion of the operating system that has been corrupted. These techniques, described below, can ensure that the system continues to be available to users, although some users may have their processes terminated in order to restore the system to a consistent state.

Both the hardware and software architectures are described in more detail below.

The Integrity S2 Hardware Architecture

The Integrity S2 has a unique hardware architecture designed to support the system's goals of high availability and high data integrity. The Integrity S2 hardware architecture is based upon Triple Modular Redundancy (TMR). The most important architectural difference between the Integrity S2 and traditional TMR architectures is that Integrity S2 uses three independently clocked CPUs.

While all three CPUs execute the same instruction stream, they do not necessarily execute the same instruction at the same time. Each CPU has its own oscillator. If one of the CPUs has an oscillator that beats slightly faster than the rest, that CPU will move ahead of the others in the instruction stream. When an interrupt occurs, all CPUs must see the interrupt at the same point in the instruction stream, lest the CPUs take different paths in processing the interrupts. Therefore, the CPUs are synchronized whenever external interrupts are presented to the CPUs.

The loose synchronization of the CPUs makes it possible to run the microprocessors at high speed. It is very difficult to lockstep multiple CPUs at the frequencies currently being used on microprocessors. Loose synchronization solves this problem without adversely impacting performance.

Each CPU has its own cache and a fast local memory out of which it will execute most of the time. The current CPU consists of a 16.67MHz R3000 MIPS microprocessor with 128KB of cache. The processors can access a somewhat slower duplicated global memory via the Reliable System Bus (RSB).

There are two self-checked voting modules that reside on the same board containing the global memory. The boards are called the TMR Controllers or TMRCs. Every time the CPUs write data into the global memory, the data is voted and is then written into both memories. A majority vote ensures that the data written into both global memory modules will be correct. A malfunctioning CPU will be outvoted and taken off-line before it can corrupt any permanent data in the system. One of the TMRCs is designated as the primary and the other is the secondary. Data is always read from the primary. A process, the primary/secondary swapper, periodically switches which of the two TMRCs is the primary.

The cache, local memory, global memory, and disk make up a memory hierarchy that is managed by the memory management software. Hierarchical memory architectures are ideal for RISC processor technology. Large caches and fast memories are required to keep RISC processors fed at rates that are fast enough to keep them from stalling. The global memory is used primarily as a fast swap device that ensures that the local memory has fast access to the active working sets of running processes.

Two I/O processors (IOPs) are connected to the global memory modules via the Reliable I/O Bus (RIOB). The IOPs provide redundant paths to I/O controllers. The IOPs also provide an interface to the NonStop V+ bus. The NonStop V+ bus is an industry standard VMEbus that has been enhanced with parity and other fault detection and isolation properties to support a more robust I/O subsystem.

Industry standard VME controllers connect through Bus Interface Modules (BIM) to the NonStop V+ buses. The BIMs allow a single controller to interface to either of the NonStop V+ buses although only one connection is active at any one time. The processor can switch a controller from one bus to another if an IOP or NonStop V+ bus fails.

Self-checking designs are used throughout the architecture, along with other methods of error detection, to increase the fault detection coverage. The identification and isolation of the errors is reported through a diagnostic subsystem. Once isolated and identified, the offending piece of hardware can then be removed from the system and a new one inserted without affecting the availability of the system.

The Integrity S2 Software Architecture

The software architecture is as unique as the hardware architecture of the Integrity S2. A great deal of creativity and invention was required in order to enhance UNIX to make it suitable for the fault-tolerant marketplace. At the same time, the integrity of the UNIX operating system source code had to be

preserved to make it possible for us to migrate to future versions of UNIX without throwing away our enhancements. The methodology we employed in our development of NonStop-UX is discussed below.

Methodology for Extending UNIX to Provide Fault Tolerance

The principal goal of the software architecture was to provide a completely standard implementation of UNIX System V on the Integrity S2, while at the same time supporting fault-tolerant operation. The operating system had to support the system goals of:

- High Availability
- Data Integrity
- User Serviceability

These features are not usually associated with UNIX. UNIX is not known for its resilience to crashes. A simple port of UNIX to the Integrity S2 hardware platform would not satisfy customer requirements.

Tandem had the benefit of surveying the results of several unsuccessful attempts to provide UNIX on a fault-tolerant platform. As a result, Tandem was able to avoid the most fatal mistake made in earlier attempts to provide a fault-tolerant UNIX platform, i.e. rewriting UNIX.

Several fledgling fault-tolerant companies have attempted to rewrite UNIX. They had come to the conclusion that UNIX was not well suited for the fault-tolerant marketplace. This is not an unfamiliar argument. There are many rewrites of UNIX available that attempt to address the real-time marketplace, the transaction processing marketplace, etc.

However, customers have failed to embrace these products for several very good reasons. Customers are extremely reluctant to purchase an operating system that is "better" than UNIX. Their skepticism stems primarily from two problems they have encountered with such solutions. The first relates to the difficulty many customers have experienced in porting an application to a non-UNIX operating system. Having to attempt a port of a large UNIX application to an operating system that deviates in any way from the UNIX interface definitions or, worse yet, that doesn't obey UNIX semantics, has dissuaded many people from purchasing operating systems that were "better" than UNIX.

Secondly, customers are anxious to gain access to the latest UNIX features and they realize that it will take longer to obtain the features if they have to depend upon the system provider to develop them. At times it seems as though the forward progress of UNIX has slowed to a crawl. The drive towards standardization requires a slow process of design by committee which impedes the

rapid development and new features required by new markets. Nevertheless, when a release does appear, customers instantly demand the features in the new release. Those who chose to emulate UNIX were not able to develop the new UNIX features fast enough to satisfy customer demands.

Therefore, Tandem decided not to implement UNIX as a layer that transformed a non-UNIX operating system into a SVID compatible programming interface. True UNIX semantics are difficult to achieve by transforming another operating system into UNIX at the system call or library level. Companies that took this approach always seemed to be behind. Their development staffs were kept busy trying to keep the interfaces up to the latest released version. Implementing new internal features was a much more difficult task than simply porting the new release of UNIX to the hardware platform. These companies lost one of the principal advantages of UNIX, namely, the technology of the UNIX operating system itself.

Moreover, many fault-tolerant systems companies have misunderstood the market's demand for UNIX to be a demand for the operating system only. This is clearly not the case. One of the prime reasons customers have switched from proprietary systems to UNIX is the availability of the UNIX software development environment. The UNIX development environment is important from several perspectives. It is superior to development environments found on many proprietary systems. It is also available on a variety of hardware platforms. Finally, companies find it relatively easy to hire and retain developers trained in that environment.

For all of the above reasons, Tandem chose to provide a UNIX implementation on Integrity S2 based on the following guidelines:

- Start from a good standard port of UNIX System V.
- Whenever possible introduce fault-tolerant features in a modular manner that will be portable across releases of the operating system.
- Ensure that none of the work to add fault tolerance will violate existing and emerging standards such as X/OPEN or POSIX.
- Ensure that no user level application software changes will be required to take advantage of fault-tolerant features.

The following sections describe a series of enhancements to UNIX which were developed in accordance with these guidelines.

Robustness Enhancements to UNIX

The UNIX kernel is a single point of failure on the system. Although there are three processors executing the instruction stream, it is

logically a single instruction stream. Therefore, a bug which would cause the system to panic or hang will cause all three of the processors to panic or hang in exactly the same way.

In order to ensure the high availability and data integrity of the system, we developed a methodology for improving the robustness of the kernel. The methodology involved multiple techniques.

Perhaps the most important technique was a rigorous quality assurance process which focused on removing defects as early as possible in the development process. The software quality assurance organization followed formalized procedures and used sophisticated tools to test and perform coverage analysis. Formal code inspections were also instituted.

Hardware assistance was provided to protect portions of memory with Write Protect RAM. Since it is not uncommon for pointers in C to be used incorrectly, the write protect RAM was used to protect kernel text from being overwritten. Our write protect hardware has a small enough granularity to be used to protect data structures as well as text, although the performance implications of write protecting data often make this approach prohibitively expensive.

If a user process is responsible for a write protect violation, then it is terminated. In general, we have followed the philosophy of terminating a process responsible for errant behavior while keeping the system available for the rest of the users.

Forward Recovery

UNIX is well known for the many calls to the *panic()* routine interspersed throughout the kernel. The *panic()* routine is often invoked after an assertion has been executed to determine if a dangerous condition exists. If something is amiss, the kernel will usually choose to invoke *panic()*, which will flush the buffer pool and shut the system down.

We began our robustness enhancements by creating a database that included information on each panic and assertion in the kernel. There are over 800 different panics and assertions in a standard System V Release 3 UNIX kernel.

We developed a multi-dimensional value function which we used to determine the priority for implementing forward recovery routines for the panics. We instrumented the kernel to determine how frequently the assertions were invoked. We then analyzed the probability that each panic might occur. These and several other metrics allowed us to prioritize the list of panics and assertions.

It is almost always possible to construct a recovery routine for an error condition. However, it may not always be wise to attempt a recovery. An assertion that is satisfied is an indication that the system has been corrupted in some way. Although it may be possible to recover from the error state by, for example, removing a corrupted element from a linked list, the original cause of the data corruption may still be present.

This presents users with a choice. In many cases, a user may value data integrity over availability. Other users may want to make the opposite choice. We created a scheme which is flexible enough to satisfy both sets of users. For the customer who would prefer to preserve data integrity by flushing the system state to disk and quickly rebooting on a fresh kernel image, an immediate panic may be invoked after the first assertion that indicates a problem.

For those with higher availability requirements, forward recovery routines will be invoked that allow the system to continue to provide services even if it means that a user process may be terminated to correct the problem. The system will then enter a probationary state where it will live for some period of time determined by the user. The system can be informed that if some number "n" of forward recoveries are attempted during the probationary period, then the system is too unstable and a shutdown will occur so that the system can be quickly rebooted.

We have also introduced a concept which we call *subscription services*. A subsystem can specify a recovery routine and request that it subscribe to it in the event of an error. Subscription services are available to any routines in the kernel. Multiple subsystem specific recovery procedures can be defined for the same type of failure condition.

Robustness Strategy

We have thought a great deal and spent a lot of time implementing a multitude of forward recovery procedures. This work will be ongoing for two reasons. First of all, we believe that the software MTBF will determine the system MTBF. Secondly, we will constantly be trying to integrate new releases of the operating system as we track the standard UNIX releases.

Because our robustness activities are so broad in scope and will be an ongoing activity for as long as we build machines with the Integrity S2 architecture, the most valuable aspect of our robustness activity is the methodology we have developed. We have developed tools that allow us to take a new UNIX release and quickly identify which new panics exist and

add these to our database.

We will continue to develop forward recovery routines and recover from errors with the same strategy. If a forward recovery routine exists, then the operating system will invoke it. If a subscription service exists, then it will be invoked. If no recovery routine exists, then the operating system will force a termination of the user process. If the operating system is not executing on behalf of a user, then it will panic.

It should be noted that we have enhanced the operating system's panic capability to ensure that data integrity is preserved should all else fail and the system has to be shut down and rebooted. When a normal panic routine is executed, an attempt is made to flush the buffer pool and update the disks to a consistent state. This goal is often difficult to achieve because the panic routine is executing on a system that may have experienced some random memory corruption.

Our panic routine uses as little of the kernel as possible when it executes. A separate driver routine runs in polled rather than interrupt mode and uses data structures set aside for this purpose. As much data as possible is write protected during the panic routine's execution. Key data structures such as the superblocks are checked for consistency before being written out to disk so that matters are compounded by writing corrupted data to the disks.

Fault Tolerance of the I/O Subsystem

The basic philosophy followed in the I/O subsystem is to have duplexed components that are checked in some way so that they can be isolated in the event of a failure. The hardware's self-checking logic is designed to detect a fault and allow isolation of a component before the fault can spread to other areas of the system. When a failure occurs an interrupt will be generated and the kernel's error recovery code will take the component off-line before it can corrupt any data. NonStop-UX is also responsible for rerouting any outstanding and subsequent I/O requests via an alternate path whenever a component has been forced off-line.

The NonStop V+ bus is a VME bus that has been modified to support the fault tolerance of the system. The NonStop V+ buses from the IOPs to the controllers have been modified to protect the data paths with the addition of parity. The buses also isolate the controllers from one another. They do this by implementing a radial addressing scheme to the VME controllers so that each of the controllers appears to be on completely separate VME bus from all the others. These changes are transparent to the controllers

hardware or firmware.

When a VME controller is added to the system, the address mapping for that controller is kept on both IOPs. If the IOP being used for access to that controller is lost, the operating system will be able to access the controller at the same address via the other IOP. This means that an IOP can fail while a device driver is in the process of accessing a controller and the driver will transparently be switched to accessing the controller through the other working IOP. This can be demonstrated on the system by starting up heavy I/O workloads and then pulling the active IOP. The BIM will switch the controller to the other NonStop V+ bus and the system will continue to operate uninterrupted.

Although most of the I/O subsystem consists of self-checked components, the off-the-shelf VME controllers used in the Integrity S2 are not designed to contain their faults via self-checking logic. It is possible, therefore, for a controller that fails and corrupts data to go undetected in the system. The Integrity S2 protects itself against such failures by generating checksums on data that is written to disk. The checksums are checked when the data is reread from disk. This technique, called end-to-end checksums, ensures that the entire path from main memory to disk and back is checked. Any failure that compromises data integrity will be identified. If the data is mirrored, then a correct version of the data can always be recovered.

Disk Device Mirroring

Mirroring is a technique for protecting disk data by writing data to two different disk drives. The two disks are often referred to as the two halves of the same mirror.

On the Integrity S2, any disk partition can be mirrored to any other disk partition of the same size. Typically, disk data will be written from memory to IOP_1, controller_1, disk_1 and simultaneously to IOP_2, controller_2, disk_2. All important data can be mirrored so that it is written to two different disk devices accessible through two completely separate paths.

Since writes to the two halves of the mirrored devices are done in parallel, the performance overhead for the mirroring is minimal. In fact, most applications will experience a performance improvement when mirrored. This non-intuitive result is due to the fact that NonStop-UX implements read optimization as well as write-mirroring. Read optimization allows the operating system to select data from either of the two identical halves of a mirrored pair. Read optimization works by selecting the least busy device or by selecting the device on which the

head is closest to the data to be read. The Integrity S2 system implements a number of different read optimization algorithms and allows any of the algorithms to be selected on a partition by partition basis.

The disk device mirroring software is implemented in a device driver independent manner so that it is available to users who wish to integrate their own VME disk controllers and add their own standard drivers.

On-Line Service

The ability to service the Integrity S2 on-line is one of the truly unique features of the system. No other UNIX system has the ability to be serviced without the loss of availability with as much ease as the Integrity S2.

To start with, the mechanical design of the system facilitates the user serviceability of the system. The system is designed so that all components and cables are accessible from the front. When the doors are opened, any of the components can be replaced by a customer without the use of any tools. The term we developed to describe the components, Customer Replaceable Unit or CRU, emphasizes the fact that trained

personnel are not needed to service the machine.

The /config File System

One of the principle requirements of the system that is an outgrowth of our focus on user serviceability is the ability to communicate information concerning the current state of the hardware and software. A pseudo file system, the /config file system, provides visibility into each hardware component and software subsystem running in the system.

The file system consists of two directory trees, a hardware directory and a software directory. The hardware directory contains files corresponding to each CRU in the system. The software directory contains files corresponding to software subsystems, e.g. System V IPC, the Powerfail/Autorestart subsystem, or performance statistics.

Status information can be obtained by "stat"ing or by sending ioctl's to the files in the file system. The real object corresponding to the file can also be operated on by opening the files and sending them ioctl's. The /config file system provides an elegant and flexible way of enhancing the user serviceability of the system in a way

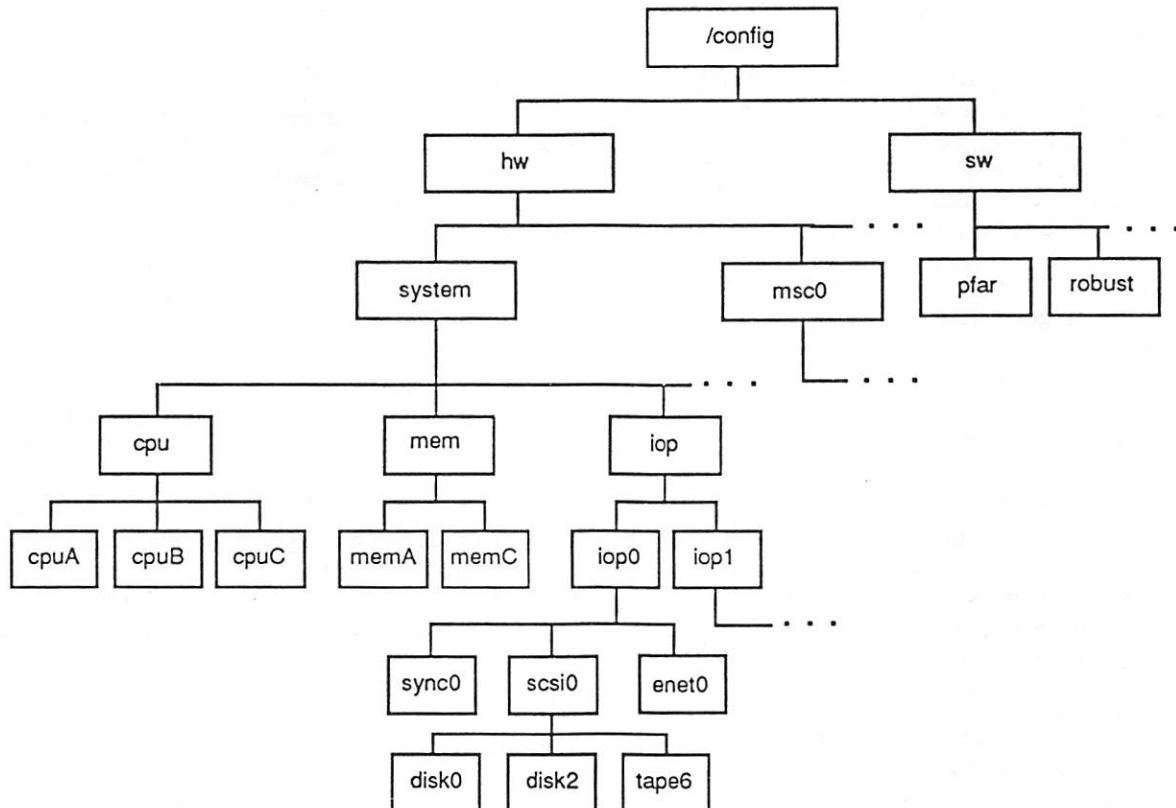


Figure 2

that is completely consistent with the UNIX model.

Reintegration

NonStop-UX supports the removal and reintegration of any of the active components within the system. If any of the boards in the system fail, repair of the system may take place without scheduled downtime; CRUs can be replaced on-line while applications are running. The reintegration process is transparent to applications and users of the system.

Reintegration of CPUs

When a CPU fails, an event log message will be generated. The administrator of the system will typically be notified of the failure via a status screen, which displays the event log messages. The administrator can choose to enable dial-out on an event by event basis. If the administrator chooses this option, the system will automatically dial-out for assistance when the event occurs.

When the new module arrives, the administrator will already have been informed of the failure and will therefore be prepared to replace the failed component. The CPU may be removed from the system at any time while it is running. The new CPU will be put in its place and it will be automatically reintegrated.

The reintegration process consists of several steps. First, the new CPU will run its Power On Self Test (POST). The other CPUs will then receive notification that the new CPU has completed its POST. The CPUs will use the DMA block copy engine to copy each local memory page out to global memory and back again. Since two of the CPUs have good data and one has bad data, the good data will replace the bad data. The CPUs will then restart from the point in the processing stream where they were previously executing. This entire process takes approximately one second on an 8MB local memory CPU.

Reintegration of TMRCs

Both the local and global memories are protected from transient soft RAM errors by a scrubber process that is part of NonStop-UX. The scrubber copies data back and forth between local and global memory. In this way, it can uncover latent parity errors. The primary/secondary swapper also periodically swaps the primary and secondary TMRC so that it is reading from both TMRCs and can uncover errors specific to one of the modes of operation.

If a hard failure occurs in a CPU or TMRC, the scrubber will not be able to correct the failure. In the case of a CPU, the CPU will be

voted offline and will need to be replaced. In the case of a global memory error, the TMRC will need to be replaced.

Once a new TMRC is installed, the good data from the other TMRC must be copied to it. This procedure happens in the background while normal processing continues. Data is simply read from the primary TMRC and written back to both. Performance can be traded off with the speed of reintegration by varying the block size of the memory copy and the time between block copies.

Reintegration of IOPs

When an IOP fails, all of the controllers on that IOP will automatically be switched via the BIM to the other IOP. Processing will continue uninterrupted.

When the IOP is replaced, the controllers that were connected to it will automatically be moved back over to it so that the system is once more balanced from both the performance and the fault tolerance perspectives.

The ability to switch controllers back and forth between the IOPs is possible because the address space used by a controller is always reserved on both IOPs. The tables which determine the address mappings of the VME controllers are kept in global memory.

Environmental Failures

Environmental failures consist principally of power failures but can include damage due to overheating, flooding, earthquake, tornadoes and other natural disasters. The most common and certainly the most tractable of these failures are the failures due to loss of power and overheating. Power failures are by far the most frequent type of environmental failure. Power failures are typically transient, lasting less than a few minutes. Even a transient power failure will cause a non-fault-tolerant system to experience loss of availability. If the system is running UNIX, chances are that data will also be lost since the disks are not kept in a consistent state.

The Integrity S2 supports continued operation through transient power failures. In the event that power is lost for longer than a few minutes, data integrity is preserved. When power is resumed, programs will automatically be restarted where they left off.

Powerfail Shutdown Procedure

Each of the cabinets in the Integrity S2 houses two bulk power supplies. These bulk power supplies normally distribute power to the CRUs throughout the cabinet. Each cabinet also houses two batteries. If external power is lost,

then the two batteries immediately and seamlessly switch in to power the system. The batteries can power the entire system for about 8 minutes.

When power is lost, the two bulk power supplies will typically transition somewhat asynchronously from a good state to a bad state. The analog nature of the power supplies will cause them to transition back and forth for a period on the order of milliseconds. A kernel powerfail process is notified of all transitions so that it can distinguish between a failing bulk and an external power loss. Once the signal is debounced by the powerfail process, the kernel will decide what action to take.

If a single bulk has failed, a message will be logged to the maintenance subsystem so that the bulk can be replaced. The system will continue to operate in the event of a bulk failure. If power has been lost, then the shutdown procedure is initiated.

The administrator has the option to mark certain processes to be killed upon power failure, but the default is for all processes to be saved so that they will be restarted at exactly the same point in the instruction stream when power is resumed. No process needs to have special code to survive a power failure.

The system will first notify all processes in the system of the power failure by signalling them. This allows customers to write applications that catch the powerfail signals and perform any clean up desired before shutdown. It also allows applications to perform initialization routines such as password verification before resuming.

The buffer pool is flushed to disk in order to ensure that the disks are in a safe and consistent state when the system is shut down. Many of the VME controllers have memory that maintains important state information. This information is copied into global memory and written out to a special powerfail partition on disk. The image of memory is then written out to the same partition.

At this point the kernel powerfail process turns off the system. The batteries have been sized so that this shutdown procedure can succeed with only one working battery. Since power failures are fairly common in most environments, we have chosen to consider them "expected events" rather than faults. We are able to handle a power failure in the presence of a single fault, even if that fault is one of the batteries. The system can also withstand two consecutive power failures, an all too common sequence of events.

Since our bulk power supplies have heat sensors within them, the Integrity S2 is able to handle overheating with exactly the same software technique that it uses to handle power failures.

Automatic Restart Procedure

When the power is restored, the controller states will be restored from the powerfail partition. The image of memory will then be restored from disk. Applications will resume processing at the same point where they left off when the power failure occurred.

A process may choose to catch the signal that is delivered when the power is restored. The application may then execute some special logic. For example, applications can invoke their own security mechanism by prompting the user for a password when the power is restored. Power failures can last for several hours and the people who were running the application may have left the area. If the application is resumed at a point where it was waiting for input from a data entry screen, the security feature could prevent an unauthorized person from entering data.

Future Directions

The Integrity S2 represents a new architecture for providing a computer system that runs UNIX and provides fault-tolerance. The main limitation of the Integrity S2 architecture is the reliance on the reliability of the operating system. The MTBF of the hardware should surpass the MTBF of the operating system by a substantial margin. The next major leap in software MTBF will probably be realized by moving to a distributed operating system. Since the industry is moving away from proprietary operating systems, this means we must wait until a distributed operating system emerges that conforms to the relevant X/OPEN and POSIX standards.

Once the UNIX operating system evolves into a distributed operating system it will become possible to return to an architecture similar to the traditional approach described at the beginning of this paper. There are benefits to a distributed operating system, such as true linear expandability, that are unrelated to fault tolerance. These benefits that are powerful enough to push the industry in their direction. The fault-tolerant marketplace will move to distributed operating system machines even more quickly because of the commensurate increase in software reliability.

References

- [1] Katzman, J.A., "A Fault-Tolerant Computing System". Tandem Computers Incorporated, Cupertino, CA, *Eleventh Hawaii*

International Conference on System Sciences, Honolulu, HI 1978.

- [2] Gray, J., "The Transaction Concept: Virtues and Limitations". Tandem Computers Incorporated, Cupertino, Texas, *Tandem Technical Report 81.3*, June 1981.
- [3] Bartlett, J.F., "A 'NonStop' Operating System". Tandem Computers Incorporated, Cupertino, CA, *Proceedings Hawaii Int. Conf. of System Sciences*, Honolulu, HI 1978. pp. 103-119
- [4] Babaoglu, O., "Fault-Tolerant Computing Based on MACH*". *ACM Operating Systems Review*, Volume 24, Number 1., January, 1990.
- [5] Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., Young, M., "Mach: A New Kernel Foundation for UNIX Development". *Proceedings of Summer Usenix*, pp. 93-112, July, 1986.
- [6] Rozier, M., Abrossimov, V., Armand, F., Bouli, I., Gien, M., Guillemart, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P., Neuhauser, W., "Chorus Distributed Operating Systems". *Technical Report CS/TR-88-7.8*, February, 1989.

Peter Norwood is currently Vice President of Research and Development at Tivoli Systems, Inc. Prior to moving to Tivoli Peter was Manager of Software Development for Tandem's Micro Products Division in Austin, Texas. The Micro Products Division is responsible for the design and development of Tandem's new Integrity line of Fault-Tolerant UNIX computers. Peter was at Tandem Computers for 7 years. Prior to that he was an MTS at Bell Laboratories in Lincroft, N.J. He has an M.S.E.E. from the University of Texas at Austin and a B.A. from Haverford College in Haverford, Pa.

DRUMS: A Distributed Statistical Server for STARS

Andy Bond, John H. Hine – Victoria University of Wellington

ABSTRACT

STARS is a mechanism for providing distributed task allocation based on the demand for and supply of resources in a heterogeneous Unix workstation environment. Previous work in this area has concentrated on either allocation of idle workstations or allocation strategies based on a simple load sharing measure. Our method of allocation attempts to model more closely the economics of resources in a typical distributed workstation environment.

A three-tiered approach is taken to solving the problem of allocation. An abstract model provides a representation of both the availability of host's resources and the resource demands of tasks. Given this common representation, we are able to use various allocation schemes to assign tasks to hosts. These allocation schemes include static, dynamic, and adaptive methods. To provide the information to the resource model, and eventually to the allocation mechanisms, a Distributed Resource Measurement Service is provided.

DRUMS is a robust and adaptive set of servers which roam the network, providing access to statistical information about hosts and tasks. Two sets of servers are used, repositories of replicated data and collectors of statistical measurements. We examine the effectiveness of using such a dynamic service for maintaining resource measurement information.

Introduction

Many current computing environments are comprised of many powerful, autonomous workstations distributed throughout the work place. The potential combined resources available in such an environment have been noted by several authors. Litzkow et al. at the University of Wisconsin have attempted to utilize the available workstations using a scheduling system called Condor[1].

From an analysis of workstation usage patterns, Litzkow notes that only 30% of workstation capacity was used. A similar 3 month study of resource use at Victoria University discovered that active workstation usage¹ peaked during the day at about 40% (see figure 2). Both studies show plenty of potential for using spare processing capacity. In addition to active workstation use, we also discovered that only a small fraction (approximately 10% of that performed by the local workstation user) of workstation processing load was performed for the benefit of remote users. This is obviously another area where performance gains can be achieved by better resource allocation.

An important method of using untapped processing capacity is through the remote allocation of tasks to hosts with spare resource capacity. This

requires knowledge of the resource usage information for all hosts in the computing environment. Such a service allows us to schedule tasks according to predicted resource requirements. We are currently undertaking work on such a system (see the STARS discussion below). As an example, we are able to rank the hosts in the environment according to some resource requirement criteria (figure 1 shows the average host ranking based upon number of users and idle cpu time over the 24-hr period). Such ranking information displays the expected daily trend in workload.

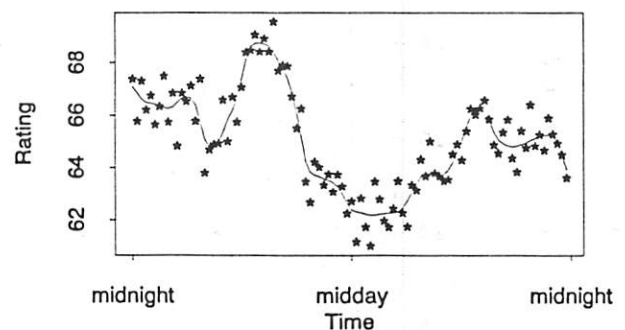


Figure 1: Average host ratings over the 24 hour weekday period for 6 weeks

¹An active workstation has a user logged onto the console and processes owned by that user use more than 10% of the available cpu time.

STARS - Shrewd Task Allocation via Resource Scheduling

Task allocation is of fundamental importance in utilizing the available resources in a modern distributed computing environment. Many methods for task allocation have been described in the literature, including allocation via heuristics[2], load balancing in NEST[3] and adaptive load balancing[4]. Many of the allocation methods use a simplistic (but none the less beneficial) view of the allocation requirements such as basing allocation requirements on a load sharing model.

STARS attempts to provide another method of allocation based upon the scheduling of resource requirements and availabilities. Methods for managing resources have been used in other areas[5] but rarely in the general Unix environment. Such scheduling requires a resource model to facilitate the comparison of predicted task resource usage with measured resource availability on hosts, available via DRUMS (see figure 3). Given such a useful comparison mechanism, it is now possible to apply both static and adaptive scheduling algorithms to provide task allocation. One of the goals of STARS is to determine the effectiveness of such an approach.

Previous Work

The approaches taken in designing DRUMS are based upon a variety of work on development of robust, distributed computations. Early work on distributed computations was undertaken at Xerox Parc in the early 1980's by Shoch and Hupp[6]. They worked on the infamous Worm programs which were robust, distributed computations designed to perform work on otherwise idle workstations. The work demonstrated the usefulness of such a programming paradigm.

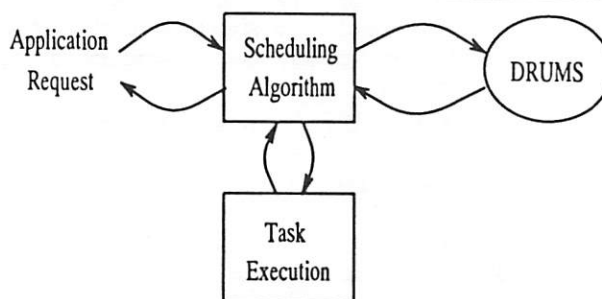


Figure 3: STARS allocation overview.

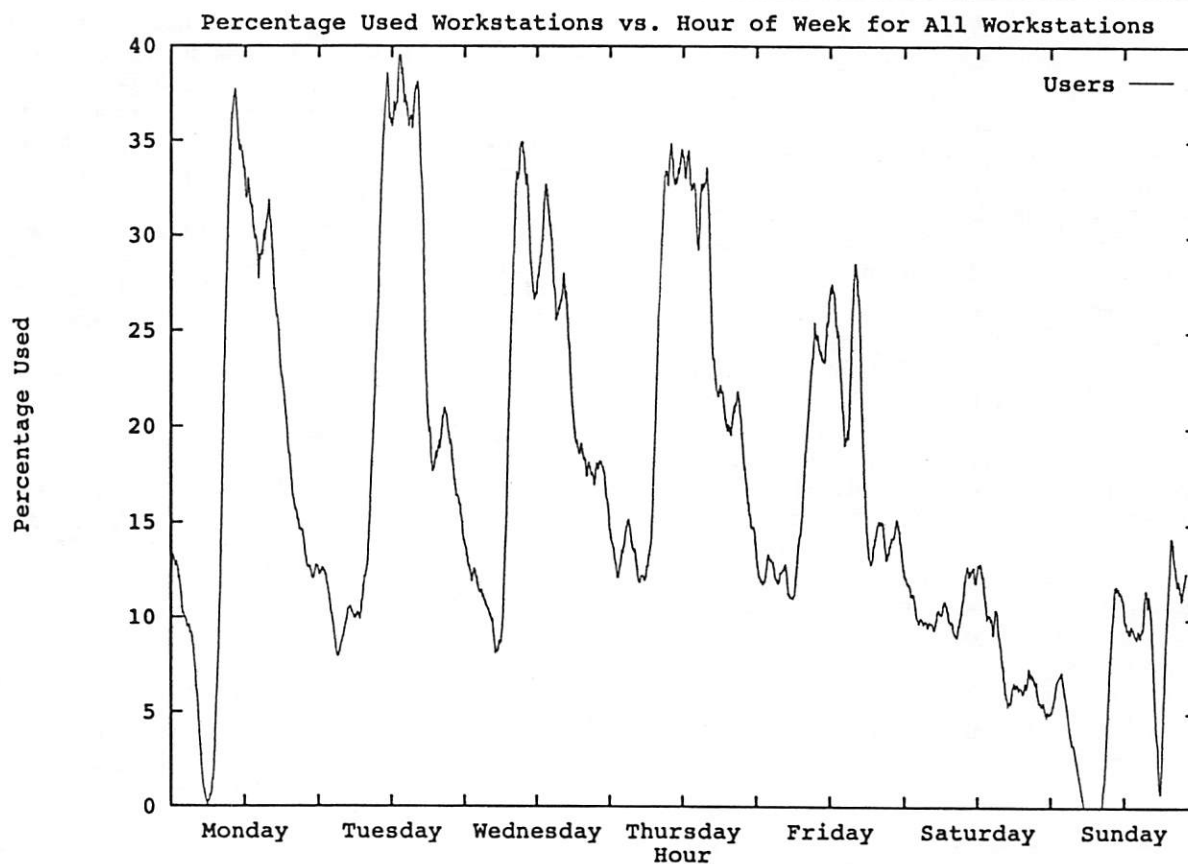


Figure 2: Percentage of active workstations over an eight week period

As a continuation of this same theme, Nichols[7] developed a mechanism called a Gypsy server which involved using idle workstations to run certain system daemons on. These Gypsy servers performed their services on idle workstations (and moved to new ones when the current workstation was reclaimed). This mechanism displayed many of the problems associated with mobile objects, namely that of binding (see a paper on object mobility in the Emerald system[8]) and service availability.

In the development of fault tolerant software, Cooper[9] describes two approaches. One is based upon the primary/standby paradigm where multiple standby wait ready to take over from a failed primary (as in ISIS[10]). The other involves the use of replicated modules. Cooper's Troupes use this mechanism to provide a robust RPC where replicated troupe members all perform the RPC client/server tasks. This redundancy provides a more robust mechanism without the overhead of replica communication as is found with the primary/standby model.

DRUMS is an amalgamation of the above techniques to provide a robust information service. Other approaches to providing information servers can be found in distributed database work[11] and distributed information services such as Grapevine[12].

The Design of DRUMS

In a typical system composed of a number of workstations, it is possible to query an individual system to retrieve a variety of measures of its current and/or recent performance. To address the problem of task allocation among workstations, or to compare the performance/load on a set of workstations we require the orthogonal view. We need to build a measurement system that holds performance information for all the hosts in the system.

DRUMS was designed to provide this performance information service. The design was lead with several goals in mind:

** The DRUMS system must be adaptive to resource availability within the network of workstations, demonstrating minimum impact on the host environment.*

A key goal of DRUMS was to minimize the effect to other users of the host systems. The service should use spare resource capacity where available, rather than impacting on work being done by users. This implies some form of dynamic configuration as described by Kramer[13].

** The server must be responsive to client requests for service.*

Since the service is to be used in the scheduling of tasks, it should provide this information in a timely fashion. Ideally the provision of service

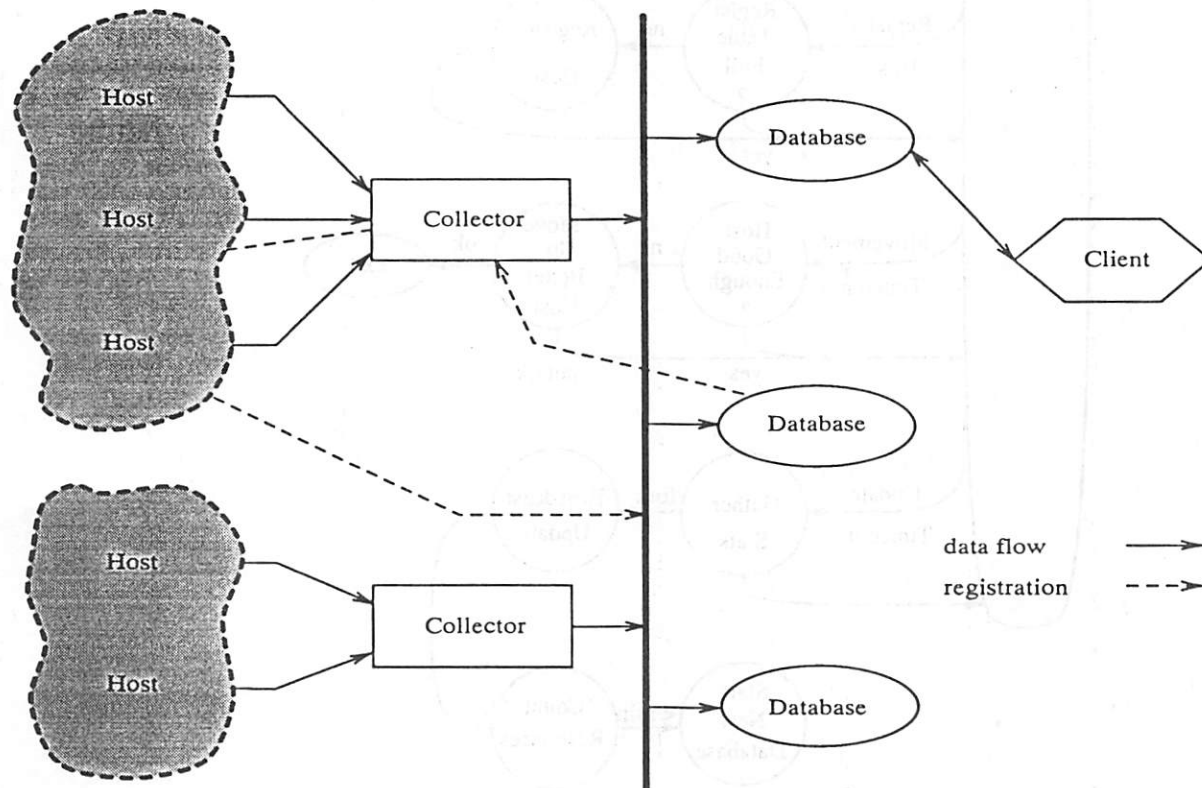


Figure 4: An Overview of DRUMS

should be independent of query load, implying a service which adapts its processing capacity to load.

** No strict coherence criteria exists for host performance data.*

The granularity of task allocation, measurement probes, and host resource measurement lag all tend to suggest that strict coherence of the performance information amongst the replicated servers is not of significant importance. It is acceptable for a server to have a less than recent version of the cpu measurement for host greta-pt.

** The system should be robust to all but the most catastrophic host and network failures*

The failure of nodes in a distributed system is not an uncommon occurrence. Major effort is put into maintaining the fault tolerance of software to partial failures. We require a service which will gracefully survive the loss of hosts and breakdown in communication.

** Simple management.*

The management required for current system services is quite enough without the addition of extra work by our service. Minimum management intervention should be required for the addition of new hosts to the system, changes in system software, etc. DRUMS should just "keep on keeping on".

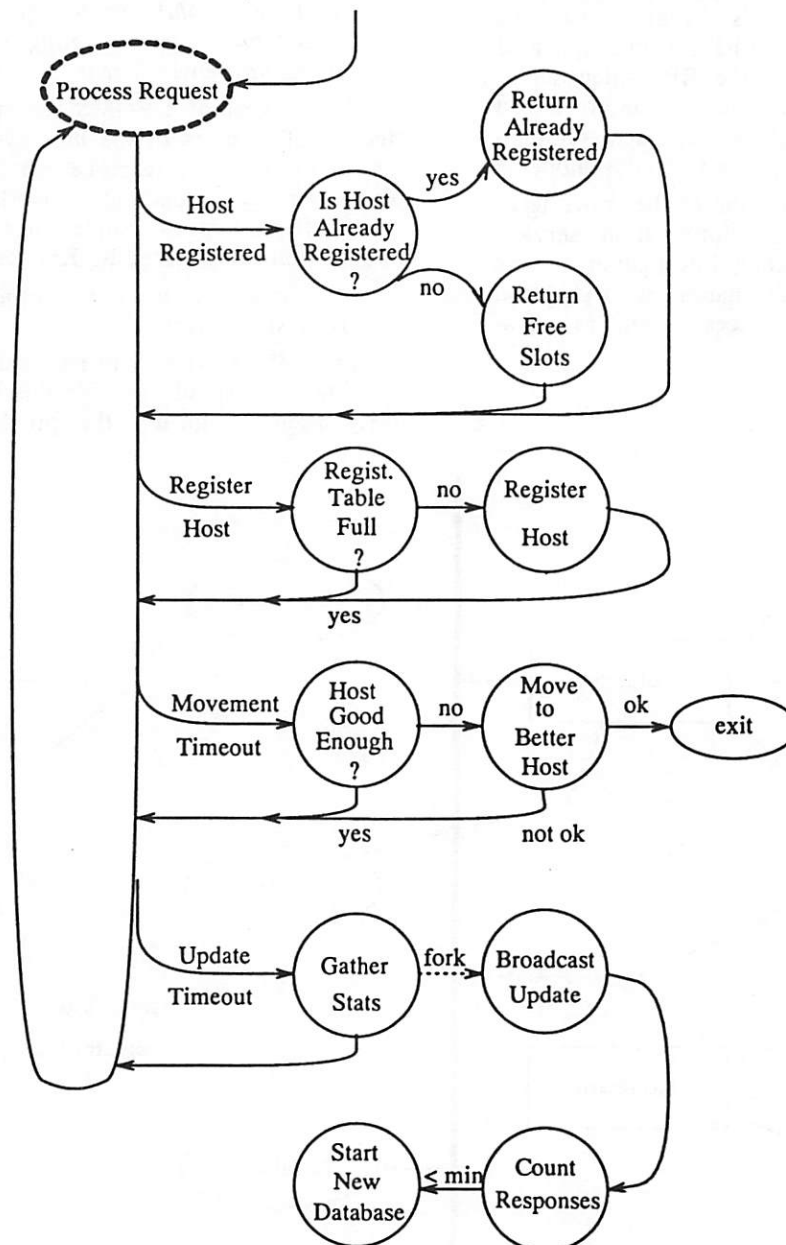


Figure 5: Event response loop for a Collector server

Overview

One major philosophy behind the design of DRUMS is the importance of replication in the design of a robust service. This paradigm is based on the observation that it is easier to recover from partial failure than from total failure. As a collective, many parts may have a higher probability of partial failure but when compared with a single module, they have less likelihood of total failure.

The service is comprised of several object classes (see figure 4).

- 1) The collection of performance information is performed by a *statistics daemon* at each host. Its purpose is to respond to requests for current performance information. This is actually implemented by a modified version of the RPC *rstatd* (Sun's RPC[14] remote statistics daemon).
- 2) The performance information is gathered, collated, and distributed by *collector* servers. These replicated servers are responsible for a unique set of hosts in the environment.
- 3) An information repository service is performed by another set of replicated servers, the *databases*. Each of these servers maintain a complete set of statistical measurements.
- 4) The final piece in the puzzle is the *interface* which provides access to the information stored at the databases.

The interesting part of the system design is the relationship amongst the databases and collector servers, along with techniques used for their management. They provide a robust implementation through a method of mutual management where the reliability maintenance is piggybacked on the inter-server communication. In conjunction with migrating servers, these techniques provide an interesting paradigm for similar distributed services.

The Organization of DRUMS

The most interesting and important work performed within DRUMS is in the database and collector servers. These servers are both implemented on top of Sun's RPC service. They provide the backbone communication and management, and due to their intertwined relationship, must be described in conjunction.

Collectors

The collectors form a replicated sub-service providing the collection and distribution of measurement statistics. Each collector in the replicated set is responsible for a (usually) unique set of hosts, called the *collector registration set*. The union of registration sets among the collectors is equivalent to the complete set of hosts in the distributed environment. As will be described later, the major disadvantage to replication with partitioned information is

the possibly of partitioned networks.

The life of a collector can be described most effectively by the logical event loop which describes its work (see figure 5).

* The registration query.

The size of the registration set for a collector is limited by time constraints on the collection and distribution of resource measurements. With the current limitations, a maximum of 24 hosts may exist in a collector's registration set. Due to this finite size, the addition of hosts to a collector's registration set is restricted.

The registration query event is used to ask a particular collector whether it is willing to add a specified host to its registration list, and thus, provide for the collection and distribution of its resource measurements. If the specified client is already in this list, the caller is informed of this, otherwise the number of free slots in the registration set is calculated and returned. This free slot information can then be used by the caller to implement a specific host registration policy (least loaded, most loaded, first answered, etc.)

* Add a client to the registration set.

Either following a registration query, or simply "off the bat", a caller can request that a client be added to the collector's registration list. Usually, a client will be added to a single collector as there is no advantage in several collectors collecting and distributing the resource measurements for the same host. If no room exists in the registration set, an error condition is returned.

The two most important services provided by the collectors are these associated with registration set maintenance. Other events handled by the service are streamlined so that most time can be dedicated to registration set maintenance. This ensures that superfluous collector servers are minimized since server availability is maximized.

* Collector movement.

At selected intervals in the life of a collector, the urge comes to start a new life somewhere else (this is internally known as a mid-life crisis). This event is triggered approximately every half hour to check on the suitability of the current host on which the collector resides. Unless the host is ranked among the top 20% of hosts capable of running a collector, the collector attempts to move somewhere else.

Collector movement may be somewhat of a misnomer as the server may actually end up leaving without being recreated elsewhere. When the collector decides to move, it attempts to re-register each host in its registration set on another collector in the environment. If each entry can be registered elsewhere, the collector is no longer needed and simply exits. This event has shown that there were too

many collectors in DRUMS and the function of movement has reduced the collector population.

A new collector must be created if:

- 1) no collector responds to the re-registration request (i.e. there are no collectors existing in the environment), or
- 2) responding collectors had no space in their registration sets.

In this case, the collector finds a well suited host with sufficient resources and starts a new collector there. This host rating method is performed in conjunction with information available from one of the databases in the DRUMS service. In practise, a new collector must be created approximately half the time and the registration list gets absorbed by the current collector population the other half. Extra collectors are created when existing collectors are busy during re-registration requests. These extra collectors then combine during the movement phase.

* *Resource measurement update timeout.*

The final event which the collector must undertake is the collection and distribution of resource measurements. In a round robin fashion, on each update event, the next registration set entry has its resource measurement information collected and

distributed to the database servers. A child process is used for this to ensure that the collector can spend as much time performing registration set maintenance (e.g., ensuring we don't get multiple host registration).

In the first example of cross server management, we see how the collectors manage the availability of database servers. Having collected the required measurement information, the collector records the number of databases which acknowledge the receipt of the updated information. If less than a minimum (currently three) number respond, the collector attempts to start a new database using a similar technique as that required for starting a collector.

Databases

The second set of replicated servers which make up DRUMS are the database servers. These data repositories contain replicated views of the current resource measurements of all hosts in the distributed environment. In contrast to a collector, on initiation a database attempts to load an initial state from some other database in the system. This gives it its first view of the world. If no database is available to provide this information, the database will have to revert to picking up the entire host

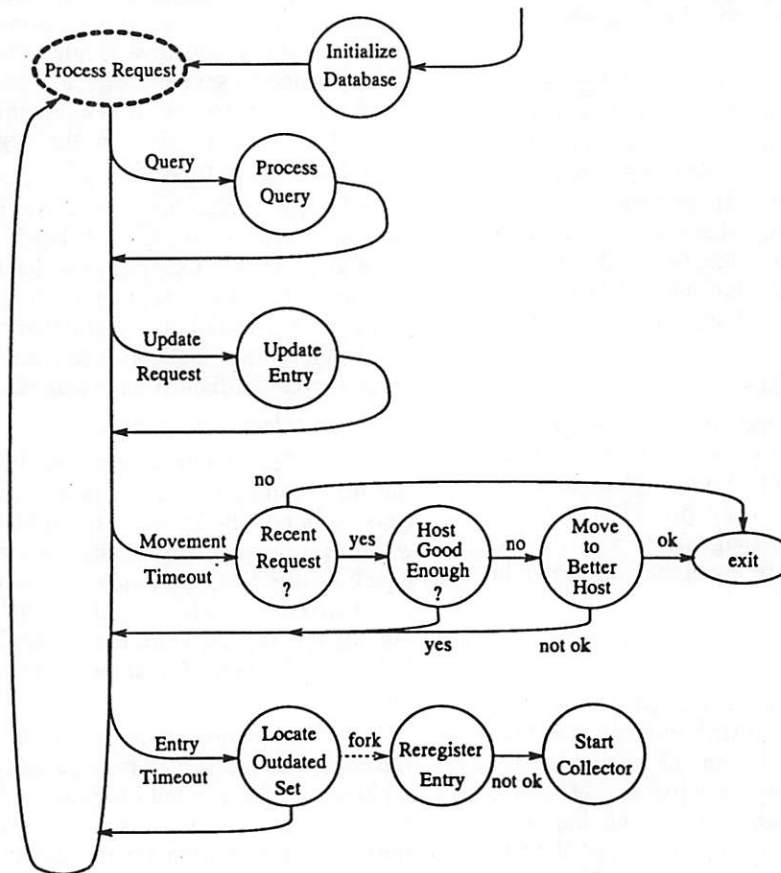


Figure 6: Event response loop for a Database server

information through updates from collectors. Since this can take a long time, it is useful that this method of information collection is a last resort and is rarely used. As with the collector, each server responds to a set of events (see figure 6).

** Process a query.*

The primary task to be performed by databases (and indeed by DRUMS itself) is to provide resource measurement information in a timely fashion to clients. As such, the databases are optimized to dedicate as much time as possible to responding to these requests. Several types of queries exist but the most important is to return a set of requested statistical information about some set of hosts.

** A collector update.*

Periodically, the database receives resource measurement updates from the collectors present in the system. Each update is recorded along with the time of receipt. Currently, the update occurs in three minutes intervals for each host recorded in the database (with 30 hosts, these updates will appear, on average, every 6 seconds).

** Database movement.*

In a similar way to collectors, databases attempt to move from hosts that aren't ranked in the top 20% of hosts for running databases. A check is made approximately every half hour, and if required, the database searches through its own resource measurement information to find a better host to move to. In order to control database numbers, each database checks to see if it recently processed a query. If not, it assumes that it isn't needed and exits. Using this method, database numbers are controlled by query rates. The intermittent requests from collectors (approximately every 30 minutes for each collector) are enough to ensure that all databases will not exit at the same time.

Unlike the collector, a database has no state to transfer (that can't be resurrected easily) nor redistribute since new databases look for an initial state on invocation. Thus, once a new database has started on another host there is no need for the old database to do anything except exit.

** Entry timeout.*

To maintain the information in the database, it is necessarily to periodically wander through the database looking for out of date resource measurement entries. These entries indicate that no update is being received from a collector for a particular host. At this point, we assume that the host indicated by the outdated entry is no longer registered with a collector. This may have been caused by a host failure where a collector was executing or communication failure between the collector and database, possibly due to a fragmented network.

At this point, we attempt to re-register the outdated host with a collector. In the case of a temporary communication failure, we may discover the host is already registered, in which case we have no problem. If the outdated entry was due to host failure, we re-register the entry and things are back to normal. An interesting situation results if the communication breakdown was due to a network partition. In this case, we end up creating two independent DRUMS systems if a database exists on each side of the break. These will then merge when the network rejoins.

If we cannot find any collector to re-register the host with, we end up starting a new collector. In this way, we have the second phase in the cross server management – the databases maintain the availability of collectors.

Implementation Issues

A new version of rstat

Rstat is the traditional statistics server used with Sun's RPC environment. It provides remote access to kernel statistics. We have extended this daemon to provide additional statistics, including:

- * information on the current console user (including idle time and existence)
- * virtual and real memory usage and availability
- * job queue lengths

Traditionally, rstat provided access to the simple count statistics kept by the kernel, e.g. disk transfer counts for each disk. These counts have been converted to rates with common related measures being averaged. As an example, we provide a single disk transfer rate. This also has the effect of creating a uniform statistics interface to callers as the same set of statistics can be found for each host in the environment.

In order to calculate these rates, rstat takes at least two measurements of the kernel statistics before responding to it's first query. Additional queries received before a timeout are answered with only one kernel query. Currently the pause between the initial two readings is 3 seconds. With this in mind, the following statistics were collected on the performance of the new server:

	Traditional	Extended
Client call (secs)	0.222	0.289
Server answer (secs)	0.148	0.195
Resident size (K)	124	204

These client/server times are the elapsed time in the call/answer. The extended server is marginally more expensive than the traditional implementation due to the extra information it deals with. The main overhead is the initial three second pause which is incurred while gathering the first statistics rates. However, after the first request, subsequent

requests can be answered at a more reasonable cost. It is a tradeoff of startup time versus the extra expense of having the daemon run continuously.

An interface to DRUMS – any

One simple interface has been developed to DRUMS. This interface is used from the command line to do static task allocation, returning the most appropriate host for the specified task. Each application is associated with two descriptions:

- 1) A logical *characteristics* expression describes the host requirements for the application. It is used as a filter for selecting appropriate hosts, e.g.

```
(HP300 || SUN3)
(hostname != embassy)
(free_virtmem > 10)
```

- 2) A set of resource *weightings* describe which resources are important for the application. These weightings are used to rank each eligible host between 0 and 100 as being appropriate for running the required application, e.g.

```
max mips 0.5 ,
max free_virtmem 0.2 ,
min load_5 0.3
```

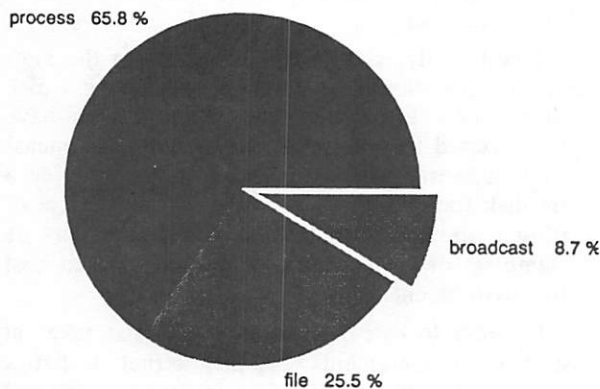


Figure 7: Cache hits for Database locating

An important issue with an interface to DRUMS is the problem of binding to one of the mobile databases. Given the infrequency with which databases move and the distribution of queries to databases, a three-tiered caching mechanism is used. Most database location cache information is taken to be cache hints (as described by Terry[15]) as opposed to guaranteed location data. Other work on binding to mobile objects includes mobility in the Emerald system [8] and location independent invocation in an RPC environment[16].

Currently, an interface to DRUMS uses a process cache for repeated calls from the same process and a file cache for repeated client calls from the same host. Figure 7 shows that more than 90% of requests are satisfied from these two cache locations.

The final resort is to search for a database using a broadcast call, performed approximately 9% of the time.

Table 1 shows the frequency of client calls that are made to the databases. Notice that the majority of use is currently by DRUMS itself. These results were recorded over a 6 week period.

User	Request Count	Request/Hour
Collector	6636	6.50
Database	6294	6.16
Other	4615	4.52

Table 1 – Interface usage by major user

Bootstrapping DRUMS

An interesting side effect of the ability of the servers to survive failure is the methods used to invoke DRUMS the first time. The obvious way is to start a number of databases and collectors, then attempt to register each host you wish to maintain statistical information on.

Another method is to start a single database and add dummy entries for all the hosts in the system. As each entry in the database times out, they will be registered with a collector (which are thus created). In turn each collector will discover there are not enough databases and create new databases. This is the proverbial “pulling it up by its bootlaces”. One potential problem with this method is the susceptibility to failure when the initial server redundancy doesn’t exist.

Experimental Evaluation

DRUMS has been in continual use for the last year, undergoing periodic upgrades to incorporate new features. During this time, it has survived operating system upgrades and many partial node failures. The current set of statistics have been recorded over a six week period.

Server Availability

Of primary importance in any distributed system is availability. The service should be available in a timely manner all the time. Impeding this goal is failure within the environment.

The service must be robust to any partial failures. Figure 8 shows the distribution of unexpected server deaths per 12 hour period. We see that the failure rate for collectors was very low but there were still periods when the collectors collapsed at anything up to six per hour. During the experimental period, the database had a periodically exercised software bug causing a large number of single failures. Looking at a continuous graph of server failure (see figure 9), we can see that these failures tend to be grouped. This is understandable due to

global system problems in a diskless environment.

One method of measuring availability is to look at the number of simultaneous servers available. Figure 10 shows the distribution of simultaneous server counts over the experimentation period. We see that the number of collectors is less variable than the number of databases. This is due to the more strict collector management via merging collectors, in contrast to the less conservative database creation. It should be noted that there were a couple of times when neither server were around. This appears to occur periodically due to log files being full (and log information being lost) and other such events. At no time during the experiment period did DRUMS require management intervention.

Server	All Hours	9 to 5
Collector	2.7	2.4
Database	3.7	3.4
Both	5.8	5.4

Table 2 – Average number of simultaneous servers

Table 2 is an attempt to determine whether the increased load (see figure 2) during working hours had an effect on the number of servers. We see that there are marginally fewer servers during these busier hours, probably due to less hosts being available to run on.

Adapting to Query Load

As a consequence of providing a service which is responsive to user requests, it was suggested that being adaptive to the query load would ensure sufficient responsiveness. The simple mechanism of collectors sending measurement updates is intended to ensure the provision of sufficient response. As the number of queries from clients increase, more of the database's time should be spent replying to queries and less to fielding updates. The collectors will notice this increase and start new databases. As a result, we would expect the number of simultaneous databases to increase as the query load increases. It should be noted, however, that since databases exit after extended periods without queries, the

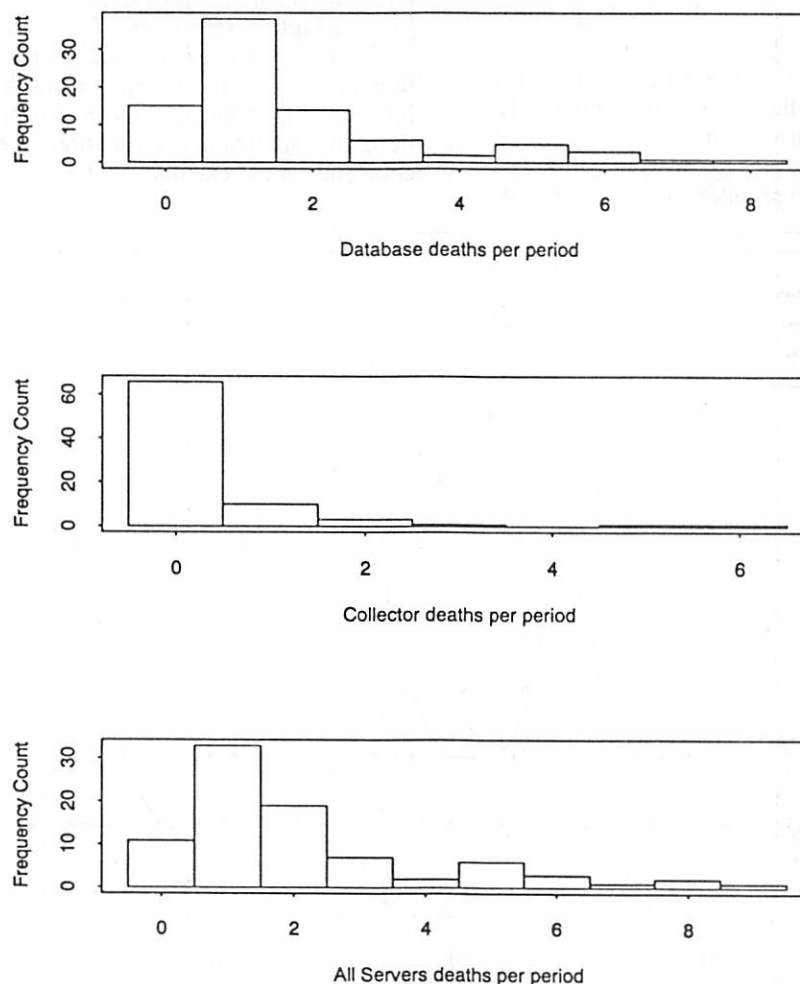


Figure 8: Unexpected server deaths over a six week period

resulting fall in databases will be slower than the corresponding decrease in query rates. The use of caching can delay this fall even further. Figure 11 shows the results of graphing such a relationship. We see that database number peaks do seem to be related to periods of increased query load. It must be remembered that other factors exist effecting database numbers, including average system load.

A Dynamic Response to the Environment

Server movement is attributable to changing host loads. As hosts become busier, less are available to run DRUMS servers but very dynamic host use may cause servers to move more often. A similar result may occur when the systems have less load. The difference between loads becomes smaller and small activities have more impact on the system load. Figure 12 shows the relative movement rates over the average weekday. The database servers move four times as frequently as the collector servers. This is caused by the additional database servers and possibly by a heavier workload performed by the database servers. Table 3 shows more server movements occur during the working hours but not significantly so.

We can also look at the server periods to get a feeling for how dynamic the servers are. Figure 13 shows the effect that the enforced retirement of underutilized databases (checked every two hours) has upon the relative server intervals. In contrast,

the collectors display a more logarithmic distribution of server intervals. As expected, table 4 shows that the servers have much shorter host occupations during the working hours.

Server	All Hours	9 to 5
Collector	4.8	6.2
Database	19.8	22.4
Both	14.5	16.7

Table 3 – Average server movements (movements/day/server)

Server	All Hours	9 to 5
Collector	3.7	1.3
Database	1.3	1.1
Both	1.6	1.1

Table 4 – Average server periods (hours/server)

Improvements and Future Work

The next phase of work in the development of DRUMS is support for the extra information required by an adaptive scheduler. DRUMS will be used as an information server for application resource prediction data as well as the current host performance information. Similar to the performance information, the application estimation doesn't have strict coherence requirements.

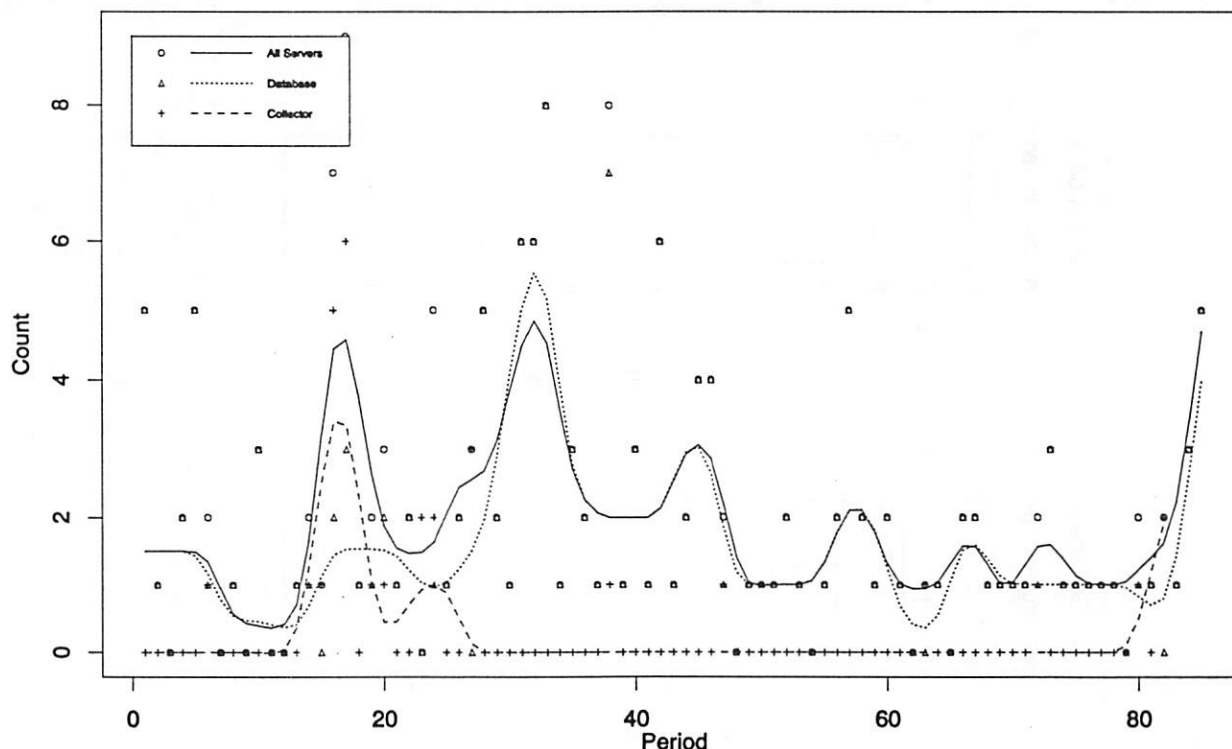


Figure 9: Unexpected server deaths per 12 hour period over a six week period

There are plans to provide multicasting to perform the distribution of resource measurement data from the collectors to the databases. This will require minimal change as they support the same basic semantics and interface.

Another potential area of use for this dynamic information service paradigm is the class of servers whose databases are built up through a caching mechanism. An obvious example is a global name server such as that proposed by Cheriton and Mann[17] in. There are requirements for replicated servers for both availability and performance, and for additional servers as the load increases. Once again, there is no requirement for strict coherence between the caches.

Conclusion

This paper has presented a distributed service for providing resource usage information to task schedulers. DRUMS is a robust mechanism utilizing two sets of reliable servers providing complementary redundancy management. The environment in which

DRUMS currently runs has resulted in a MTBF for individual servers of 5 to 6 hours. The service is adaptive to query load and robust to all but the most disastrous failures. It uses server mobility to ensure minimum impact to other computing users. We have described the design of the mechanism and extensive performance analysis. The design of DRUMS should be applicable to other data collection applications and services with minimal coherency requirements.

References

1. Litzkow, Michael J. and Livny, Miron and Mutka, Matt W., "Condor - A Hunter of Idle Workstations," *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 104-111 IEEE, (June, 1988).
2. Efe, Kemal, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, pp. 50-56 (June, 1982).
3. Ezzat, Ahmed K., "Load Balancing in NEST: A Network of Workstations," *1986 Fall Joint*

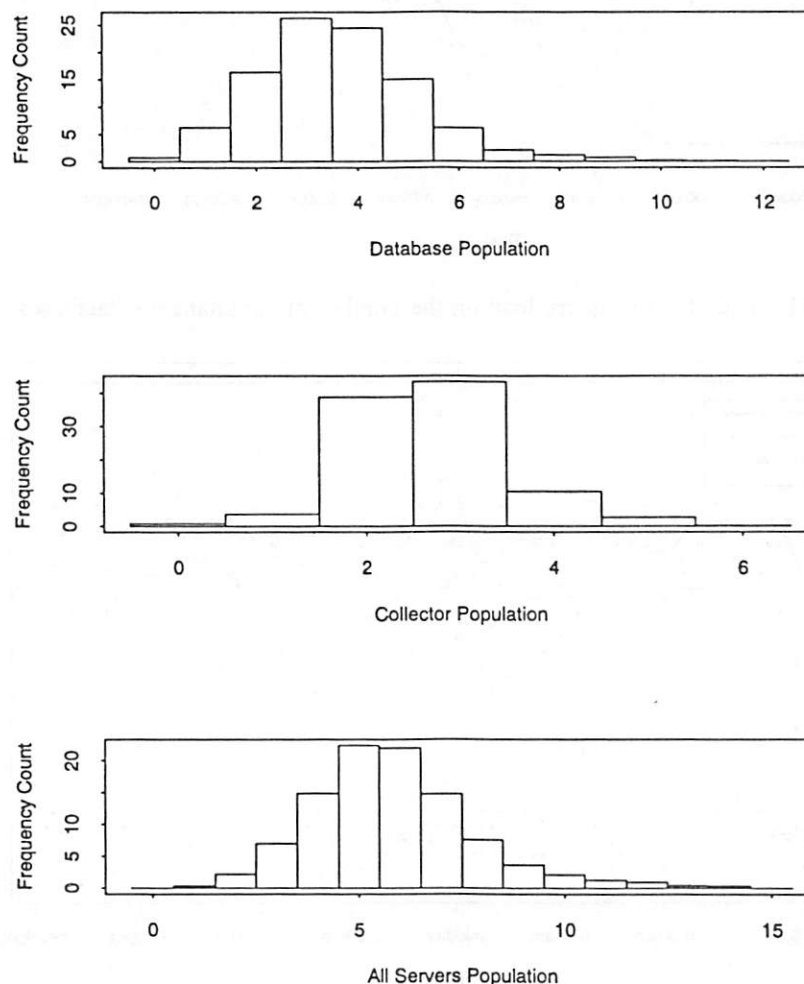


Figure 10: Frequency distributions of simultaneous server numbers a six week period

Conference of the IEEE/Association of Computing Machinery, pp. 1138-1149 IEEE, (November, 1986).

4. Eager, Derek L. and Lazowska, Edward D. and Zahorjan, John, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering* SE-12(5) pp. 662-675 (May, 1986).
5. Craft, Daniel H., "Resource Management in a Decentralized System," *Proceedings of the Ninth Association of Computing Machinery Symposium on Operating System Principles*, pp. 11-19 Association of Computing Machinery, (October, 1983).
6. Shoch, John F. and Hupp, Jon A., "The 'Worm' Programs - Early Experience with a Distributed Computation," *Communications of the Association of Computing Machinery* 25(3) pp. 182-180 (March, 1982).
7. Nichols, David A., "Using Idle Workstations in a Shared Computing Environment," *Proceedings of the Eleventh Association of Computing Machinery Symposium on Operating System Principles*, pp. 5-12 Association of Computing Machinery, (October, 1983).

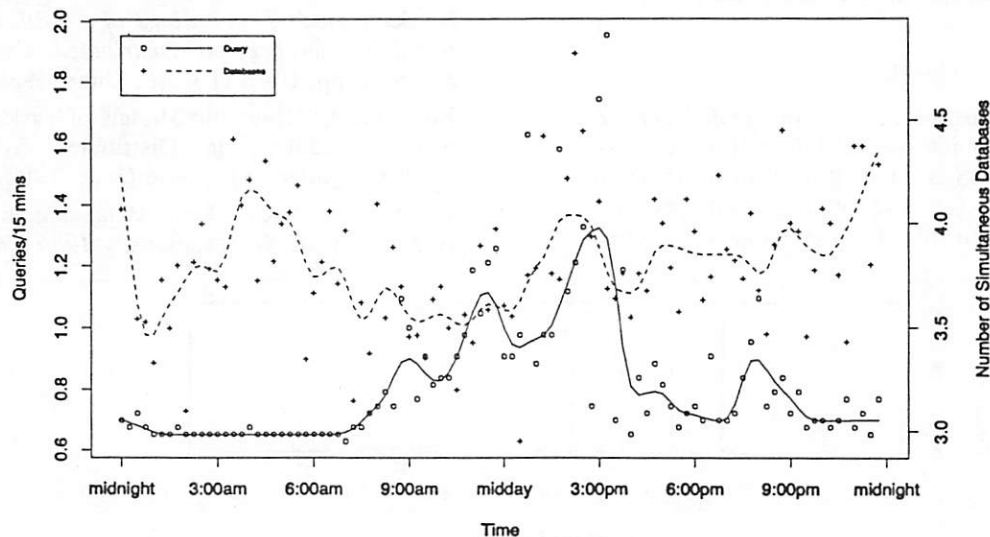


Figure 11: The effect of query load on the number of simultaneous databases

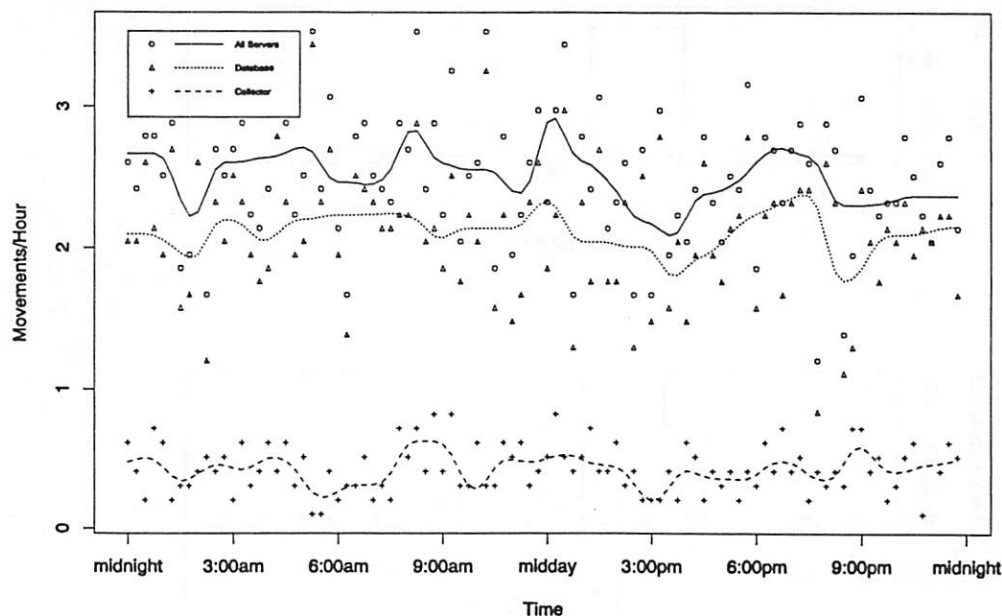


Figure 12: Number of server movements per hour

- Machinery, (1987).
8. Jul, Eric and Levy, Henry and Hutchinson, Norman and Black, Andrew, "Fine-Grained Mobility in the Emerald System," *Association of Computing Machinery Transactions on Computer Systems* 6(1) pp. 109-133 (February, 1988).
 9. Cooper, Eric C., "Replicated Distributed Programs," *Proceedings of the Tenth Association of Computing Machinery Symposium on Operating Systems Principles*, pp. 63-78 Association of Computing Machinery, (December, 1985).
 10. Birman, Kenneth P., "Replication and Fault-Tolerance in the ISIS System," *Operating Systems Review* 19(5) pp. 79-86 (December, 1985).
 11. Sarin, Sunil K. and Lynch, Nancy A., "Discarding Obsolete Information in a Replicated Database System," *IEEE Transactions on Software Engineering* SE-13(1) pp. 39-47 (January, 1987).
 12. Birrell, Andrew D. and Levin, Roy and Needham, Roger M. and Schroeder, Michael D., "Grapevine: An Exercise in Distributed Computing," *Communications of the Association of Computing Machinery* 25(4) pp. 260-274 (April, 1982).
 13. Kramer, Jeff and Magee, Jeff, "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering* SE-11(4) pp. 424-436 (April, 1985).
 14. Sun Microsystems/Mt Xinu, *Remote Procedure Programming Guide, 4.0 Edition*. November, 1988.
 15. Terry, Douglas B., "Caching Hints in Distributed Systems," *IEEE Transactions on Software Engineering* SE-13(1) pp. 48-54 (January, 1987).
 16. Black, Andrew P. and Artsy, Yeshayahu, "Implementing location independent invocation," *IEEE Transactions on Parallel and Distributed Systems* 1(1) pp. 107-119 (January, 1990).
 17. David R. Cheriton and Timothy P. Mann,

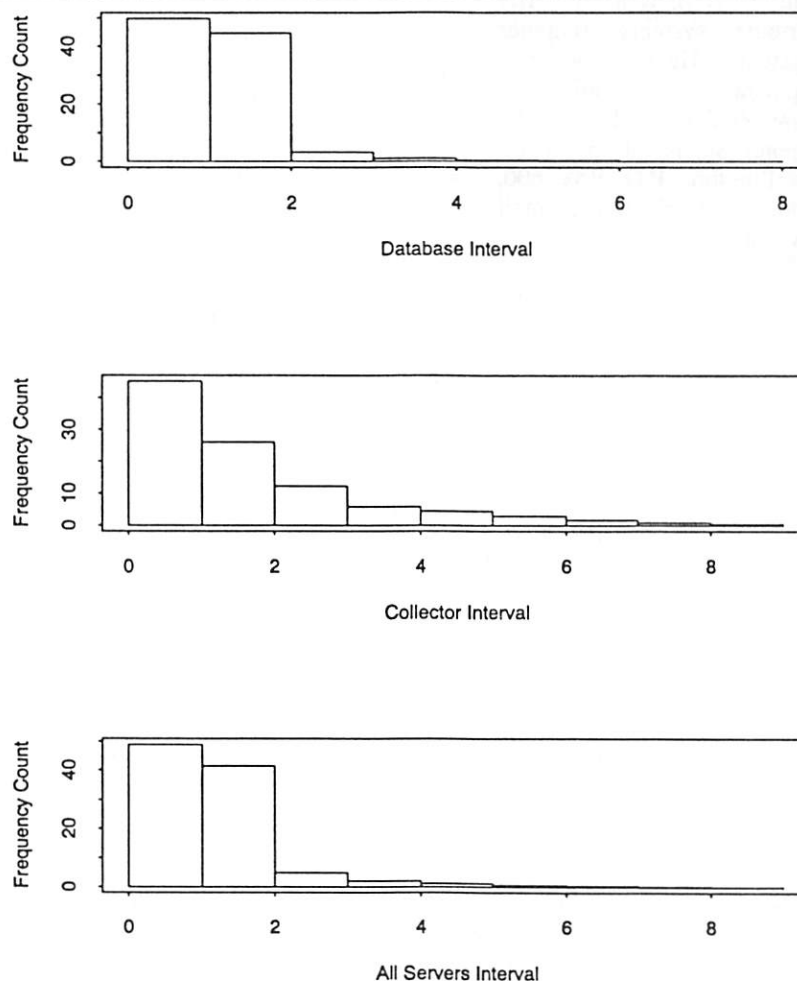


Figure 13: Frequency distribution of server intervals (in hours)

"Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance," *Association of Computing Machinery Transactions on Computer Systems* 7(2) pp. 147-183 (May, 1989).

Andy Bond is currently a PhD student at Victoria University of Wellington. In 1987, he completed an Honours degree in Computer Science at Victoria and has since been researching task allocation in distributed systems. His research interests include distributed and multi-computer systems. He can be reached via mail at Computer Science Department, Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand. His electronic mail address is andy@comp.vuw.ac.nz.

John Hine is Professor of Computer Science and Head of Department at Victoria University in Wellington, New Zealand. Before joining Victoria in 1977 he was on the staff at the University of Newcastle-upon-Tyne, England. John is currently a member of the board of UNIFORM NZ and editor of *USR*, a New Zealand Unix newsletter. His research interests are distributed systems, computer networks and operating systems. He has also been involved in the development of academic and research networks in New Zealand. He can be reached via mail at Computer Science Department, Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand. His electronic mail address is hine@comp.vuw.ac.nz.

Experience Building a Process Migration Subsystem for UNIX

Dan Freedman – University of Calgary

ABSTRACT

Process migration has been explored for a number of years as a means of achieving performance improvement in distributed computer systems through load balancing. Due to the complexities inherent in the technique, most implementations have relied on the ability of an operating system to provide location-transparent addressing of computing resources (devices, memory, and so forth), so that these resources may be accessed easily by processes after they have been migrated. This paper describes a process migration subsystem which takes a different approach. It takes advantage of whatever location-transparent addressing facilities the operating system provides, however where not available it gives processes the opportunity to make their own arrangements. Under this implementation, process migration is not transparent, but rather relies on active co-operation between processes and the migration subsystem itself. Using this co-operative strategy, it has been possible to develop an effective process migration scheme for UNIX without modifications to the UNIX kernel. Although applications wishing to take advantage of the process migration subsystem require some modifications, for many programs these are straightforward. Under this subsystem, process migration is most transparent to processes which spend a large amount of time processing data in memory, and which use little in the way of operating system services. Some support is provided for processes which have to terminate and then re-establish connections with the operating system during process migration.

Introduction and Background

Process migration, the moving of running processes from one computer to another, is a technique for automatically putting idle computing resources in a computer network to work. It also allows the releasing of resources when they are required by higher priority processes. Conceptually, it involves taking processes from overworked machines, and moving them to machines which are under-utilized, or which are better suited in some way to running them. Thus the resources of idle machines, such as workstations assigned to co-workers who are not currently using them, can be "borrowed" and (significantly) given back when needed by other users. The ability to give back a particular computing resource such as a workstation while still continuing with process execution is what differentiates process migration from remote execution mechanisms (such as rlogin and rsh under UNIX, or the Butler system [NIC90]) which are simpler to implement. Under remote execution, processes must be paused (preventing further computation for the time being) or killed (negating the processing already completed) in order to give back a workstation's resources. Under process migration, resources are given back by moving processes to other computers where computation can continue. A process need only be paused when no suitable computer can be found for it to migrate to.

A secondary application of process migration is as a tool to increase fault tolerance [CHO83]. When it is known that a computer will be unavailable (for example due to scheduled down-time), process migration allows processes running on that computer to be transferred to another machine. When the computer comes back on-line, processes can be migrated back again as appropriate. Of course, resources used by the migrating processes (such as files) may also need to be migrated if they are to remain available.

The concept of process migration is straightforward, but its realization is complex. Processes interact with the operating system, other processes, memory, co-processors, files, devices and other resources, and thus have associated with them many items of location-dependent state information (file descriptors, device locks, IPC connections possibly with data in transit, and so on) used when accessing these resources [LEF89]. This state information must be migrated with the process, and translated as necessary to ensure its continued validity in a new environment. This state information is often dispersed throughout the process, and may not be easily identifiable by the operating system. For example, UNIX file descriptors are simply integers, and may be copied to and stored in any memory location. To compound the problem, some state information is stored away from the process itself, perhaps in databases maintained by the kernel. If the kernel is extensible, state information may end

up in locations unknown to both the original kernel and the process. UNIX device drivers (which are linked in with the kernel) can exhibit this characteristic. Unless care is taken with the distribution of process-specific state information, situations can develop where neither the operating system nor the processes themselves have the ability to collect and translate all process-related state information.

One solution is to have the operating system provide location-transparent addressing of computing resources such that no distinction is made between local and remote resource accesses [POW83] [RAS81] [THE89]. Under this kind of scheme, state information remains valid regardless of the location from which it is used. When location transparent addressing is applied in both directions (from resources to processes as well as from processes to resources), the fact that some state information may not have been migrated (because neither the kernel nor the process knew of its existence) becomes irrelevant to the continuing correct operation of a migrated process, although there may be performance implications.

Location-transparent addressing is a conceptually clean and attractive solution, providing programmers with what appears to be a large integrated computer system rather than a network of loosely-related smaller systems. However, this solution is not (yet) offered by major operating system vendors. One reason is that implicit in a location-transparent addressing scheme is a relatively tight coupling between the databases maintained by the operating system kernels throughout the network(s). Typically, UNIX systems running on networks of workstations provide only a loose coupling between kernels, and it is not a simple matter to increase the level of integration while maintaining reasonable performance.

It is possible to implement forms of process migration without modifying the operating system kernel, although there is inevitably some loss of transparency. As a Master's thesis project, the author has implemented a non-kernel process migration subsystem (PMS) for UNIX which concentrates on preserving the memory image of a process across migrations, but pays little attention to ensuring the continuation of operating system services (such as sockets, files, and devices) in use by the process. The PMS is thus well suited to processes which do not use much in the way of operating system services, but instead spend most of their time manipulating data in memory. For example ray-tracing, image analysis, seismic interpretation, and some simulation programs typically operate by initially reading in data from a file, processing the information in memory for an hour or so without using any special system services, and then writing out the results to another file. As long as process migration does not occur in the short initial and final stages of

execution, no maintenance of operating system services across process migrations is necessary.

The PMS migrates processes among all participating computers in an attempt to balance the cpu load average across the network. No machine will be left idle if other machines are heavily loaded. Processes are migrated away from computers where interactive users are working so that background processing can proceed without interfering with interactive use. The PMS can co-exist with normal (non-migrating) processes on a workstation, since only processes which register with the PMS will be considered eligible for migration. Process migration under the PMS is non-preemptive. Processes receive migration instructions from the PMS process scheduler, and may choose when and/or whether to follow them. Processes which need to break and then re-establish connections with the operating system or other processes when process migration occurs can use pre- and post- migration hook function pointers which can be assigned to subroutines in the process. The PMS will call these subroutines immediately before and immediately after it migrates a process. This hook mechanism broadens the range of processes to which the PMS can be applied. It is conceivable to create a library of support routines which would ease (although not totally eliminate) the burden of maintaining links with the outside world across process migrations. The initial implementation is oriented toward cpu-intensive programs which interact infrequently with the outside world, and does not yet provide these libraries.

The rest of this paper describes the process migration subsystem, the scheduling system developed to decide when and to where processes should be migrated, and experience gained while applying the subsystem.

Process Migration Subsystem Overview

Any implementation of process migration must contain two fundamental components: a scheduler and a migration mechanism. The scheduler is responsible for deciding on which computer each process should execute. It must collect information from the computers onto which processes can be migrated, and from processes which wish to be migrated. It must interpret this information, coming up with a schedule with an assignment of processes to computers. It must then cause the migration mechanism to initiate migrations for those processes which are to be moved. The migration mechanism is responsible for pausing a process, collecting up its state information, transporting the information to the computer on which execution is to continue, and resuming the newly-migrated process. If process migration is to be truly transparent, the state information must be translated as necessary in order to maintain its validity across the migration. For example, file descriptors must continue to refer to

the same files in the same way after migration as they did before. In some systems, an additional responsibility of forwarding to the new computer any communications which may arrive at the old computer for the migrated process may be assigned to the migration mechanism.

There are two scheduling goals for the PMS scheduler. First, if any foreground (interactive) use is being made of a computer, all migratable processes should be scheduled to other computers, or if no suitable computer can be found, all such processes should be paused. This goal encourages people to allow background processing to occur on their own personal workstations when they are not being interactively used, since as soon as interactive users begin to work, they will command absolute priority over their machines. Workstations should not appear to be "bogged down" by background tasks when interactive users are trying to work. The second scheduling goal is to balance the cpu load average of all machines on the network (except those being used interactively) by migrating processes from relatively busy machines to relatively idle ones. Under UNIX, the cpu load average represents the average number of processes which are ready to use the cpu (as opposed to waiting for i/o or a signal to occur for example). Thus balancing the load averages ensures that no machines lay idle while others are overloaded. This scheduling algorithm differs from more traditional ones (such as those developed from [STO77]), which require in-depth *a priori* analyses of process execution and communication patterns. Since the PMS is oriented toward compute-intensive independent tasks, the amount of inter-

process communication a process makes is assumed to be minimal. Similarly, it is assumed that processes will continue to require the same amount of computing resources on all machines.

The PMS consists of two major components, the *migrationd* daemons, and the *libmigrate* run-time libraries. Scheduling, coordination, and process transportation services are provided by the migration daemons, *migrationd*. One *migrationd* runs on each computer in the network. *Libmigrate* consists of a run-time library of routines which interface the process they are linked in with to the *migrationd* daemons, accepting, interpreting, and acting upon instructions and requests from both. Each process wishing to take advantage of process migration has the *libmigrate* library linked in with it. This organization is depicted in Figure 1.

Figure 2 illustrates the steps that are performed in order to migrate a process from one computer to another. Each *migrationd* is responsible for deciding where the processes currently executing on its computer should be migrated to in order to improve performance. This scheduling is based on load statistics gathered from its own computer and broadcast by other *migrationd* daemons operating on computers with no interactive users. If there is a computer on the network with a load average significantly lower than the local load average (and which has no interactive users), *migrationd* decides that one of its local processes should be migrated to that computer. "Significantly lower" is a settable parameter designed to avoid useless migrations between computers whose load average differs by a trivial amount. To avoid many processes being

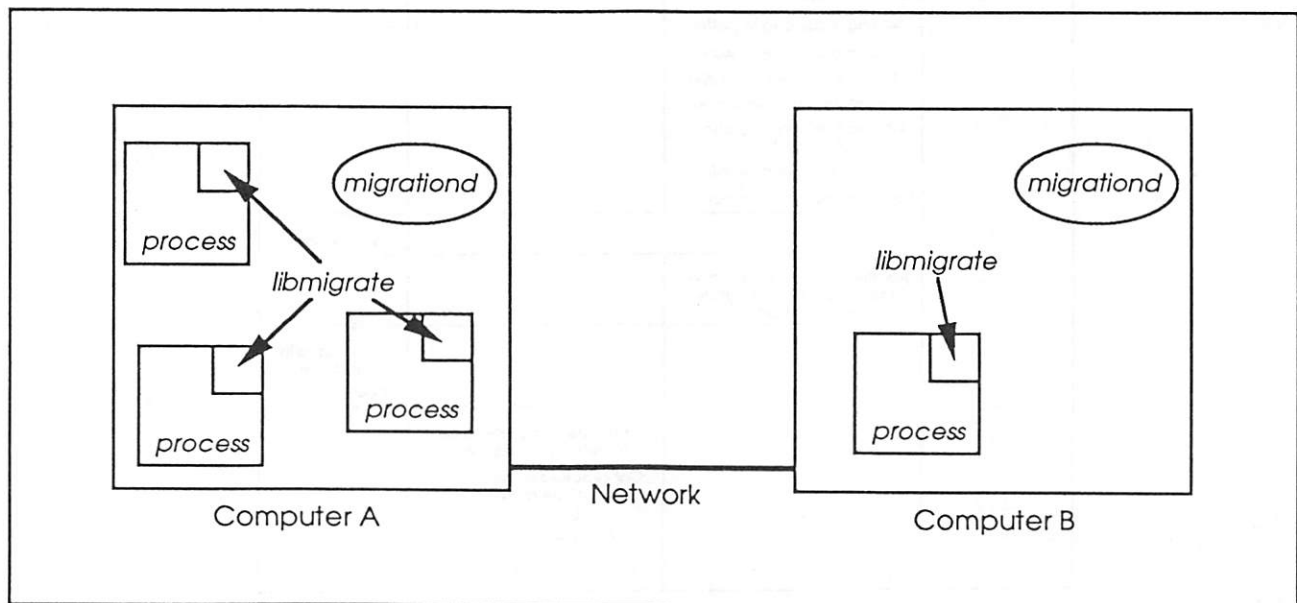


Figure 1: Major components of the Process Migration Subsystem

simultaneously migrated to a lightly-loaded computer, a reservation system is used. Once a *migrationd* decides that one of its processes should be migrated to some new destination computer, it tries to make a reservation for the process with the destination *migrationd*. The destination *migrationd* can refuse the reservation for a variety of reasons: if too many processes have already had reservations made for them, if the load average has climbed such that the destination computer no longer seems attractive, or if interactive users have begun work. If a reservation is refused, the *migrationd* which tried to make it can try another computer if one exists with a sufficiently low load.

If the reservation is successfully made, the *migrationd* advises the process that it should migrate to the destination. This advice is received by one of the routines in *libmigrate*, which copies it for later

use, and sets a flag indicating that advice has been received. Under PMS, processes wishing to take advantage of migration must periodically call a subroutine *migrate_if_necessary* which checks to see if advice has been received from the local *migrationd*. If it has, it attempts to follow the advice and migrate to the indicated destination. Since some time may have elapsed between receiving the advice and proceeding with the migration, the *migrate_if_necessary* routine confirms its reservation with the destination *migrationd*. Again, the reservation could be refused, in which case *migrate_if_necessary* returns, and the process continues as normal. If the reservation is accepted however, *migrate_if_necessary* collects the memory image of the process, and transmits it to the destination *migrationd*, which saves it as an executable file. When executed by the destination *migrationd*, the

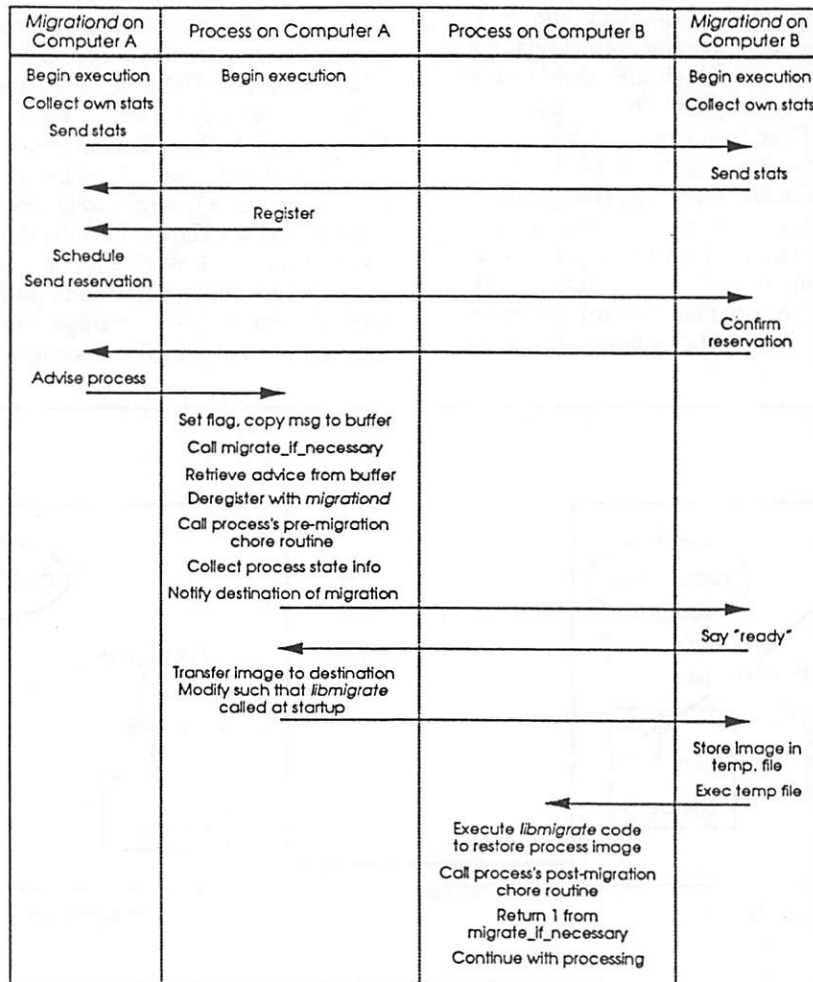


Figure 2: Timeline of scheduling and executing the migration of a process

process memory image (stack and heap) are restored, and execution continues as if the *migrate_if_necessary* subroutine were returning normally. The return value of *migrate_if_necessary* indicates whether or not process migration has occurred.

Applying the Process Migration Subsystem

A ray tracer developed in C++ by the Graphics-land group at the University of Calgary was chosen as a test for the PMS because it involved very few interactions with the operating system, spending almost all of its typical 1 hour run-time operating on data in memory. The only state-involving operating system interactions are file-related. At the beginning of the run, the ray tracer reads in a data file containing the frame to be processed. Once per scan line, output is written to another data file. Apart from the file i/o, all the ray tracer does is interact with objects in memory.

```

begin ray_tracer
    open input file
    read in data
    close input file

    init libmigrate library
    register with migrationd

    for each scan line
        migrate_if_necessary
        ray trace the scan line
        open output file
        move to end of file
        write output for scan line
        close output file
    end
end

```

Figure 3: A ray tracer using the PMS

Figure 3 represents the modifications made to the ray tracer in order to make it eligible for process migration. After the initial reading in of data, the *libmigrate* library is initialized, and the process registered with *migrationd*. Once per scan-line, the *migrate_if_necessary* routine is called to begin process migration if it has been scheduled by

migrationd. The ray tracer had originally opened the output file at the start of the run, and closed it at the end, simply writing out data once per scan line. Since file descriptors are not maintained by the PMS across process migrations, the ray tracer was modified to open and close the output file each time a scan line is written. Since output is relatively infrequent, this solution is satisfactory. A less wasteful approach would be to only re-open the output file when process migration occurs. Either the return value from *migrate_if_necessary* (1 if migration occurred, 0 otherwise) or the post-migration hook functions could be used for this purpose. The makefile for the ray tracer was modified to link in the *libmigrate* library. No other modifications to the ray tracer were necessary.

The program needed to be run once for each of 21 frames to be produced in the test run. Each run was independent of the others, so each frame could be ray-traced in parallel to the others, using all idle workstations to spread the load. A network of 9 Sun Sparcstations was used for the experiment, which was run during the day with several users working intermittently on some of the workstations. The workstations all ran the NFS file system, eliminating a worry about locating the processes on a particular workstation when file interactions were necessary. A *migrationd* daemon was started on each workstation, and left to run for a minute or two to allow time for statistics to be circulated. The ray tracer processes were then started on one of the workstations. The local *migrationd* interpreted the typing of commands on the workstation console as interactive use, and so began to schedule migrations to other machines for the ray tracers. This initial migration-intensive period lasted about 6 minutes, after which time all of the ray tracers were spread among the other available machines. If the processes were manually started on different machines, this initial period could be avoided.

The test run lasted about 4 hours, during which time 63 process migrations occurred. Since not all of the frames required equal amounts of processing time, some ray tracers finished long before others. As a result, the load average on some workstations dropped to well below that of some others. When this happened, the *migrations* on the busy computers tried to schedule migrations for some of their own processes onto the less-busy machines. Due to the reservation system, only some of them were successful, resulting in a few migrations over the course of the run. The majority of the migrations occurred as a result of users beginning to use workstations interactively. When this happened, the *migrations* on those workstations migrated their processes to other machines. Users reported that they could tell when process migration was occurring by watching their network transceivers' flashing LEDs, and by listening for disk activity. Other than an initial

period of a few seconds at the beginning of interactive use where severe memory paging was experienced, users were not inconvenienced by the background use of their workstations. Since migration is i/o rather than cpu intensive, little degradation in performance was evident while processes were migrated.

Summary, and Further Work

Process migration is a technique to access appropriate computing resources at appropriate times, and to free those resources when it is no longer appropriate to use them. It has been possible to set up an ad-hoc process migration subsystem that does not require modifications to the operating system kernel, although as a result it is not completely transparent. Only the memory image of a process is migrated. The validity of state information provided by the operating system such as file descriptors, process ids, and socket addresses, is not maintained across migrations. As a result, the system is best suited to those processes which do not interact frequently with the operating system, but rather spend most of their time manipulating data stored in memory.

An ad-hoc distributed scheduling mechanism allocates processes to computers based on two goals. Processes must be migrated away from computers with interactive users, and the cpu load average of all computers on the network should be more-or-less equalized. A reservation system is used to resolve race conditions caused by many computers trying to schedule processes onto a lightly-loaded machine. The process migration subsystem has been successfully applied to a ray-tracing application with very little effort. Users reported little inconvenience when the process migration subsystem was run on their workstations. Only a few seconds of severely degraded performance were reported when users began to interactively use workstations with migratable processes on them. Once the initial paging-intensive period passed, processes migrate quietly to other computers without severely degrading local performance.

Remote execution (using UNIX commands like `rlogin` and `rsh`) is considerably simpler to implement than process migration, but does not allow computing resources to be released in mid-execution without completely stopping a process. Process migration allows the process to continue elsewhere, assuming an appropriate location for it to run exists. The automatic load balancing facility of a process migration system also frees users from having to worry about where processes should or should not be executing.

The process migration subsystem could be improved in a number of ways. More support could be provided for processes which interact frequently

with the operating system. The Condor process migration system [LITZ88] provides more (although not total) support of this kind. Condor however, requires that all operating system interactions be referred (by the Condor run time library) back to the computer on which the process was started, so the original computer is never completely freed. Theimer, Lantz, and Cheriton describe an optimization made to process migration under the V system whereby a process continues to run while its memory image is migrated to its new location [THE89]. Those parts of the image which are modified while migration is occurring are re-copied. The process is only paused when the set of changed memory pages becomes sufficiently small. This considerably reduces the amount of time large processes spend being paused while being migrated. Reduced latency helps avoid deadlock or timeout problems that might otherwise occur.

Scheduling in the current implementation is based upon heuristics and a number of carefully-chosen scheduling parameters. Problematic situations exist for this scheduler, yet they are easy to fix on a case by case basis by adjusting the parameters. For industrial settings where similar programs are executed every day on similar datasets, this may prove to be satisfactory. However a more comprehensive scheduling algorithm is desirable. Scheduling algorithms for distributed systems have been published, but are compute intensive and often rely on information about process execution patterns that is difficult to collect. An adaptive version of the currently implemented scheduler may be easier to implement than the traditional algorithms, and may satisfactorily avoid thrashing problems.

The process migration subsystem described here clearly leaves a lot to be desired in terms of transparency but, without modifying the operating system kernel, there will inevitably be facilities provided by the operating system which are not properly maintained across process migrations. However, for users who run compute-intensive tasks which interact little with the operating system or with other processes, the subsystem provides a viable means of gaining access to unused computing resources.

Acknowledgements

Many thanks to Dave Hankinson and Theo Deraadt for making available their encyclopaedic knowledge of the SunOS operating system, and to Dave Jevans for providing the ray tracer used in the test run.

References

- [CHO83] T. C. K. Chou and J. A. Abraham, "Load Redistribution Under Failure in Distributed Systems," *IEEE Transactions on Computers*, Vol. C-32 No. 9. September 1983.

- [LEF89] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, "The Design and Implementation of the 4.3BSD UNIX Operating System," Addison Wesley Publishing Company, Reading, Mass. 1989.
- [LITZ88] M. Litzkow, M. Livny, and M. Mutka, "Condor -- A hunter of Idle Workstations," *Proc. 8th Intl. Conf. on Distributed Computing Systems*, San Jose, California. June 1988.
- [NIC90] D. A. Nichols, "Multiprocessing in a Network of Workstations," Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. 1990.
- [POW83] M. L. Powell and B. P. Miller, "Process Migration in DEMOS/MP," *ACM Operating Systems Review*, Vol. 17 No. 5. October 1983.
- [RAS81] R. F. Rashid and G. G. Robertson, "Accent: A Communication Oriented Network Operating System," *Proc. Eighth Symposium on Operating System Principles*. December 1981.
- [STO77] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Transactions on Software Engineering*, Vol. SE-3 No. 1. January 1977.
- [THE89] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," *ACM Operating Systems Review*, Vol. 25 No. 5. December 1989.

Dan Freedman is currently completing an MSc. degree in Computer Science at the University of Calgary. He is primarily interested in operating systems and programming environments, particularly in providing convenient and efficient access to remotely-located computing resources. His Master's thesis presents and analyzes the process migration subsystem described in this paper. During the last 6 years, he has worked part-time as a research programmer and consultant/problem solver at the University of Calgary. Since 1988, he has also been a director at Freedman Sharp and Associates, a Calgary-based consulting firm specializing in systems software and administration. When not working with computers, Freedman can most often be found playing or listening to Jazz and contemporary pop music. He performs regularly in Calgary on piano and keyboards, and is currently preparing a solo CD. He is also a Ham radio operator (VE6DFM).



Dan Freedman can be reached at the Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta,

Canada. T2N 1N4. Telephone: (403) 220-6015.
Fax: (403) 284-4707. Email: dan@cpsc.ucalgary.ca.

A Modular Architecture for Distributed Transaction Processing

Michael Wayne Young, Dean S. Thompson, Elliot Jaffe – Transarc Corporation

ABSTRACT

The prevalence of Unix systems has made parallelism and distribution an attractive solution for transaction processing. The affordability of Unix systems makes them attractive both as intelligent front-end processors and as serious computation engines. Unix systems have encouraged the construction of small tools that can be combined to solve larger problems. Modular system components with open interfaces simplify the construction of efficient, reliable, distributed systems. Standard operating systems interfaces are part of the solution. Open, modular transaction processing interfaces are another important part.

Existing transaction systems fail to define clean interfaces for transaction management, communication, logging, locking, and recovery. Their communication protocols, recovery algorithms, and transaction management algorithms are deeply inter-related. This complicates the implementation and makes interoperability with other systems more difficult.

The Transarc TP Toolkit is a highly modular set of transaction processing components that simplifies distributed systems programming. At the core of the Toolkit is a distributed transaction management service, which coordinates the commitment of transactions that involve multiple applications. This transaction management service defines strict interfaces to communication and recovery components. The transaction management component is responsible solely for ensuring a consistent transaction outcome. The communication component is responsible solely for the interaction with remote applications. It may use any communication protocol, naming scheme, security model, or programming interface -- in particular, the underlying protocol does not need to contain the details of transaction management. The recovery component is responsible solely for maintaining local persistent resources, and may choose its own locking or logging techniques. The Transarc TP Toolkit provides efficient implementations of these transaction management, communication, recovery, locking, and logging services, many of them in small program libraries, on a variety of operating systems. The Toolkit also includes C language support that makes it easy to write both transactional applications and recoverable server programs.

Introduction

The Transarc TP Toolkit is a modular collection of components that provide efficient, portable solutions for distributed transaction processing systems. The availability of portable operating systems with standard interfaces on inexpensive hardware has made distributed computing very attractive. The basic *transaction* paradigm has proved useful for building reliable applications. Unfortunately, existing transaction processing system interfaces are proprietary, and their architectures are monolithic. The TP Toolkit defines modular interfaces for the building blocks for distributed transactional applications: commitment management, communication, logging, locking, and recovery. This architecture permits an efficient implementation that outperforms many traditional architectures.

Standardization and distributed computing

Operating system standardization has opened the door for new distributed applications in several ways. The availability of Unix operating systems on a wide range of hardware platforms has created a fiercely competitive marketplace. The cost of hardware alone encourages migration from single massive mainframes to collections of smaller, less expensive machines. System interface standards permit further savings through portable software.

The TP Toolkit is a layer on top of existing standards. The Base Development Environment (BDE) component of the TP Toolkit provides a portable operating system level interface, including such constructs as threads of control, synchronization, memory management, and file I/O. The BDE implementation is layered on other standard interfaces (e.g., OSF/1, SVID, Posix).

Transaction model

The TP Toolkit provides a very rich transaction model that forms the framework for building reliable, efficient, parallel and distributed applications. The basic transaction computing model, described by Gray¹ [Gray78], allows programmers to define *transactions* that complete or fail as a unit. The nested transaction model, described by Moss² [Moss85], allows large transactions to be broken down into smaller *subtransactions* for improved failure recovery and isolation. The TP Toolkit includes full nested transaction support without the constraints on parallelism found in previous systems.

Transaction systems eliminate many of the problems introduced by parallel and distributed execution. Programmers define collections of work, called *transactions*, that represent consistent changes to distributed data. A transaction can include work in several processes on one machine or on several machines, possibly in parallel, as specified by the programmer. The transaction system ensures that each transaction is:

- executed atomically, meaning it either completes or fails as a unit;
- isolated from other transactions, meaning its changes are not visible to other transactions until it completes; and,
- durable, meaning that once it completes, its changes persist despite hardware failures.

Programmers need not be concerned with the effects of failures during the execution of a transaction -- the transaction system undoes the partial effects of a transaction when any failure occurs.

Nested transactions are a mechanism for breaking down large transactions into smaller transactions that each have the same atomicity and isolation properties. Transactions can be nested arbitrarily, forming trees of related work. Failure of a subtransaction causes its partial effects to be undone, but does not cause the enclosing transaction to fail; the subtransaction can be retried, possibly using alternative resources that have not failed. Subtransactions can be safely run in parallel; the transaction system treats them as independent units that must be isolated until they complete. Failure of a transaction causes its constituent subtransactions to be undone as well; therefore, a subtransaction is not completely durable until the transaction at the root of its tree completes.

Previous systems have provided weaker mechanisms for breaking down large transactions, or have restricted the use of parallelism among nested

transactions. Some systems provide a *savepoint* mechanism [IBM84] for establishing firewalls beyond which failures do not cause work to be undone; these firewalls are similar to a single level of nesting. Other systems that include full subtransaction support have required the use of nested transactions in order to do distributed³ [Liskov87] or parallel⁴ [Eppin91] work. The TP Toolkit provides full nesting but overcomes these restrictions.

Additional TP modularity required

In order to make transactions available to a wider audience, transaction systems must achieve greater modularity among their components, and must make the interfaces to those components public. Existing transactions systems are often monolithic and proprietary. This makes it difficult impossible to take advantage of new components (e.g., communication systems, relational databases, programming languages) from other software vendors, and makes incremental improvements to old components harder. Where systems have published interfaces, they have failed to sufficiently separate the functions performed by each component. A more modular architecture can result in components that are better specified, functionally independent, and more portable.

There have been many attempts at establishing practical standards for transaction management, but none have been a complete or modular solution:

- The LU6.2⁵ [IBM85] and ISO TP⁶ [ISO90] protocols embed transaction management in the underlying communication system. This combination of transaction management and communication makes it nearly impossible to adapt either the commitment protocol or the basic communication interface without affecting the other. More importantly, these systems make the use of other communication paradigms within a transaction difficult. Lastly, the combination of communication and transaction protocol makes these specifications a nightmare to understand and implement.
- The currently proposed X/Open XA+ interface for communication provides a procedural interface to embed transaction management in the communication system. By using a procedural interface rather than an explicit network

³Argus is a programming language environment that incorporates transaction management.

⁴Camelot was a Carnegie Mellon research project in transaction management.

⁵LU6.2 (sync level 2) is a connection-oriented protocol that includes a two-phase commit protocol. It was the first widely used, commercially available transaction management protocol.

⁶The ISO TP protocol is a recent standard that is very similar to the LU6.2 protocol.

¹Gray formalized the concept of a transaction and its basic guarantees.

²Moss described a synchronization and recovery model for nesting transactions.

protocol, XA+ succeeds in letting more than one communication module participate in a transaction. However, each module must still understand transaction management details; the only gain over the specific protocols is that a communication module can incorporate those details into its protocol as it sees fit. Changes to the transaction management system are still impossible without changing each communication module.

- The currently proposed X/Open XA interface⁷ [XOPEN90] for resource managers (e.g., databases) requires that communication be embedded in the resource managers instead. The interface treats resources used by a process as though they were part of that process. A remote resource (e.g., a separate database process) must provide stubs for inclusion in the application process; those stubs must handle the communication with the remote resource. This architecture results in several configuration problems: the stubs must accommodate the threading model imposed by the XA interface; the naming of a particular instance of a resource (e.g., the specific database process) is difficult; and, a copy of the stubs code (the interface to actual resource processes) must be linked into the transaction commitment manager process, requiring reconfiguration to use a new resource.

These design problems often lead to increased communication or logging, as we will show later.

An architecture that more properly breaks down transaction management has several basic software engineering advantages:

- Better specification. The system is easier to understand and use because its interface definitions are visible and well defined.
- Proper assignment of responsibility. Each component can be built independently, resulting in a simpler implementation. An implementation (e.g., a data transmission scheme) can be changed without adversely affecting other components (e.g., the commitment coordination algorithm).
- Portability. Each component can be used with a variety of other implementations.

The TP Toolkit Architecture

The TP Toolkit breaks down transaction management into several components, each with a specific responsibility:

- *Commitment coordination.* The Distributed Transaction Service (TRAN) ensures that all

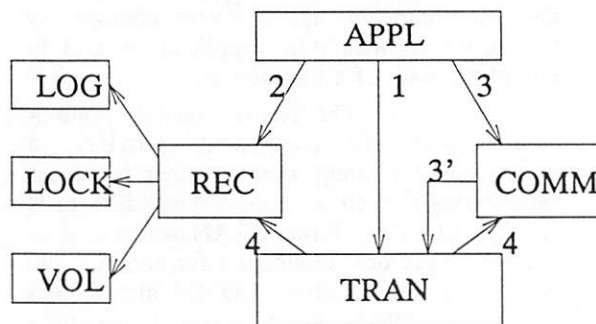
participants in a transaction agree on its outcome (the *atomicity* property). TRAN provides functions to allow applications to begin and end (commit or abort) a transaction, and to find out when important events occur during the lifetime of a transaction. TRAN uses a well-defined procedural interface to each of the communication and recovery components to coordinate with other applications and to record the state of a transaction.

- *Communication.* The Transactional Communication (COMM) component provides a mechanism for doing work in other processes (*distribution*), such as remote procedure calls (RPC). COMM informs TRAN when flow of control leaves one application for another, and when that flow returns. COMM also implements the TRAN interface for transmitting opaque data to a remote TRAN service.
- *Recovery.* The Recovery Service (REC) component controls access to transactionally persistent data, and recovers that data after failures. The REC component is only present in processes that actually maintain permanent storage; client applications that access recoverable storage in other processes do so through a COMM component. REC uses basic storage, logging, and locking services in its implementation.
 - *Storage.* The Volume Service (VOL) component provides a simple permanent storage abstraction. VOL is responsible for the correlation between logical volumes of data and the physical media used to store them.
 - *Logging.* The Log Service (LOG) component provides an efficient write-once storage abstraction that is the basis for the *durability* property.
 - *Locking.* The Lock Service (LOCK) component provides transactional synchronization among accesses to shared data (the *isolation* property).
- *System services.* The Base Development Environment (BDE) component provides basic system services, such as memory management, threads of control, and I/O. The BDE layer provides added portability across different types of operating systems. Among Unix platforms that conform to standard system interfaces, the BDE implementation is highly portable.
- *Application logic.* The application (APPL) component defines transactions (via TRAN), makes remote procedure calls (via COMM), and accesses local recoverable storage (via REC) as appropriate. The Transactional-C language component provides simple

⁷XA is a procedural interface between resource managers (e.g., databases) and a transaction manager.

constructs for doing transactional work in the C programming language. These constructs isolate programmers from the details in each of the TP Toolkit component interfaces.

The interactions in a normal transaction are shown in Figure 1.



1. An application program calls TRAN to begin/end/abort transactions.
2. An application program (e.g., a database server) can call its REC module to modify permanent storage. REC accesses the permanent storage (VOL), acquires transactional locks (LOCK) and logs changes (LOG).
3. An application program (e.g., a client) can call its COMM module (e.g., an RPC stub or library) to invoke remote work. COMM informs TRAN that the transaction is spreading and picks up state information to carry along.
4. TRAN makes calls to REC and COMM during commit processing to log state changes and deliver messages.

Figure 1: TP Toolkit Architecture.

Several features set this architecture apart from other systems:

- The interfaces are defined in terms of procedures. Many of the interfaces can be implemented either: as program libraries for efficiency, or for ease of integration with existing programs; or, as remote procedure calls for maximum isolation.
- The TP Toolkit architecture does not distinguish between "applications" and "resource managers" as other systems do. A resource manager is merely an application that contains a recovery service component. Client applications call other server applications using a communication service. That communication service may be one specifically tailored for that application server (as would be done in an XA resource manager stub) or a general-purpose one (e.g., NCS, ONC, ISO TP).
- Each interface makes minimal assumptions about other components, specifically to allow a variety of implementations. Transarc provides

an implementation of each TP Toolkit component, but they are not required. A wide variety of implementations of communication protocols and recovery strategies can be used. Other implementations are encouraged.

The Distributed Transaction Service

The transaction service (TRAN) is the central component in the management of a distributed transaction. It provides functions for the application to begin a transaction, and to ask that it be committed (its effects be made durable) or aborted (its effects be undone). It records the spread of a transaction to other applications, and drives the commitment protocol, making *upcalls* to the communication and recovery components. It allows modules to register *callback* functions to be executed at particular points in the lifetime of a transaction (for example, to allow them to do last-minute work). Finally, it provides advanced functions for controlling the commitment process (e.g., preparing early for commit, choosing commit coordinators, selecting optimizations).

The Transarc implementation of this component is done as a program library. This makes the basic functions for beginning and ending transactions and for spreading work to other applications extremely fast. It also eliminates the extra messages and logging present in systems that use a separate transaction manager process.

Communication

The communication service (COMM) component is responsible for informing TRAN when remote work is invoked on behalf of a transaction, and for providing a basic transport for messages between TRAN components in different applications. COMM makes calls to TRAN when a transaction spreads, for example during a remote procedure call; TRAN returns opaque transaction state data to be transmitted with the remote invocation. The remote COMM calls TRAN to turn that opaque data back into a transaction identifier. During commit processing, TRAN makes upcalls to COMM to send other opaque data to a remote TRAN component. TRAN and COMM perform orthogonal functions: TRAN does not understand how communication takes place; COMM does not understand transaction state or the commitment algorithm.

One Transarc implementation of a COMM service is based on the NCS remote procedure call system. RPC definitions are preprocessed to add parameters for transmitting TRAN state; stubs are generated to automatically call TRAN to fill those parameters. An additional RPC interface (with one function) is defined for passing messages from one TRAN component to another. This COMM implementation was a very straightforward layer; we expect implementations for other RPC systems to be

easy as well.

Another Transarc COMM implementation is based on the LU6.2 protocol. The design of this COMM is much more complicated, requiring use of several of the advanced TRAN features in order to allow a remote (non-Transarc) site to use the LU6.2 primitives for committing a transaction. Emulating the LU6.2 protocol also requires maintaining and logging additional conversation state that is not necessary in other COMM modules. This type of COMM implementation is possible for interoperating with many existing applications, but it highlights the difficulties present when commitment protocols are combined with with basic communication protocols.

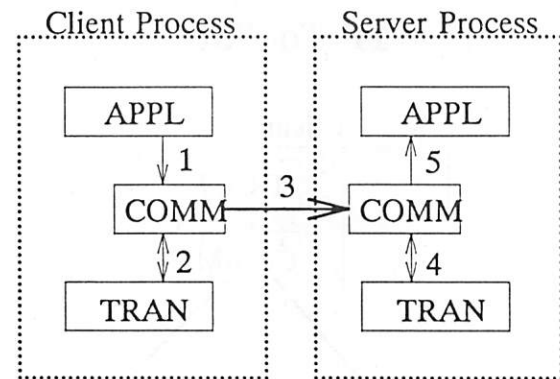
Recovery

The recovery component (REC) is responsible for managing the permanent storage affected by transactions, and for providing logging support to TRAN for transaction state information. A typical REC component (e.g., in a database server) buffers pages of its permanent storage in memory and writes descriptions of changes made to those pages to a *log*. If a transaction aborts, REC uses the log to undo its changes. To *prepare* to commit, REC forces the log to disk so that those changes can be reapplied after a subsequent crash. TRAN directs REC to prepare, and then to commit or abort work for a transaction, as agreed upon by all participants in the transaction. TRAN provides opaque data for REC to include with its log; after a crash, REC replays this data to TRAN to enable TRAN to recover the state of the transaction. This TRAN data can be written along with corresponding REC log records, eliminating extra log operations.

Transarc provides a general-purpose REC implementation that can be used to build higher-level storage abstractions. This REC implementation provides buffer management for generic volumes of data, supports general operation-based descriptions of changes to that data, coordinates with transactional locking services, and performs the appropriate logging. A server, such as a filesystem or database, can layer its data abstractions on the basic storage provided by this REC.

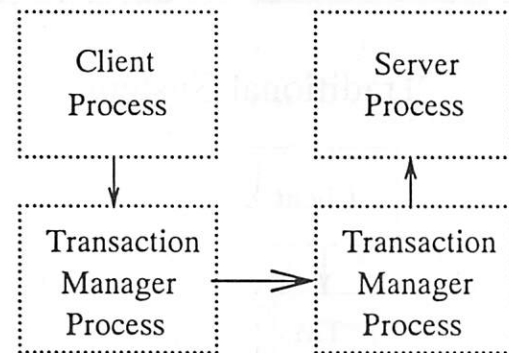
Existing servers already perform many of the functions that the Transarc REC implementation provides. These servers only need to support the interface between TRAN and REC. TRAN issues two-phase commitment upcalls (prepare and commit) to REC, regardless whether TRAN uses another (optimized or more robust) protocol between applications. TRAN includes opaque data to be written with the server's prepare log record. The server returns this data to TRAN when it reads its log during restart. TRAN does not impose restrictions on how the server logs its changes, how it prepares, commits or aborts, or how the server processes its log after a crash.

TP Toolkit



Message is sent directly, without extra local messages or context switches

Traditional System



Use of a proxy process requires local messages for remote (network) invocation.

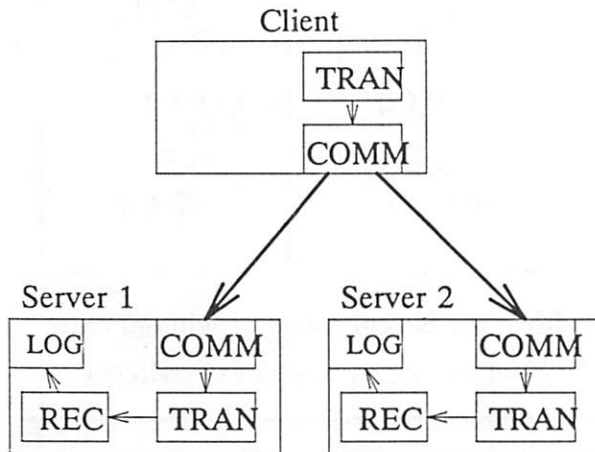
Figure 2: Remote Work Invocation Flow

Operational Comparison

The TP Toolkit architecture and the ability to implement its components as libraries results in reductions in the important primitive transaction system costs -- messages and log records written. The basic functions for beginning and ending a transaction are merely local procedure calls to the TRAN library; in other systems, these functions are often system calls or remote procedure calls to other separate system components. Likewise, local procedure calls are used to acquire transaction state when spreading a transaction to another participant; messages do not need to be routed through a separate transaction manager service. During commit processing, the elimination of a separate

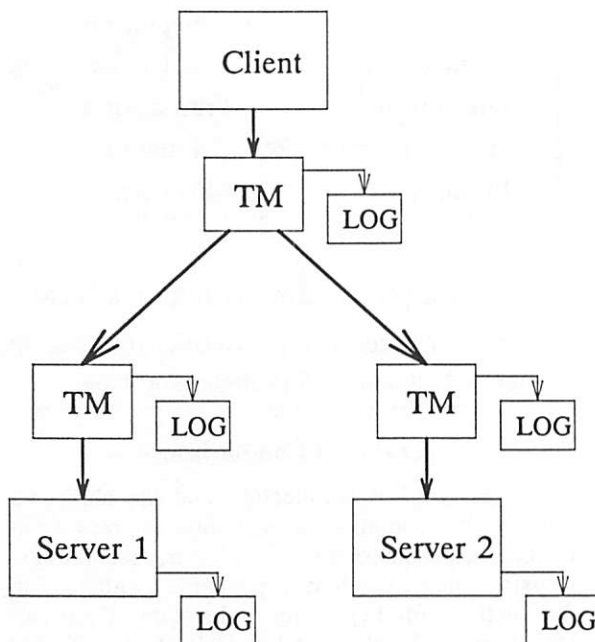
transaction service saves both messages and synchronous log writes.

TP Toolkit



No local messages; two log writes

Traditional System



Three local messages; five log writes

Figure 3: Commit Processing Flow

Other systems have taken different measures to improve performance:

- integrating the commitment protocol with the

communication system. The lack of modularity in this solution complicates the communication system.

- sharing memory between the transaction manager and separate applications in the local case. This results in different mechanisms for local and remote access, and may result in reduced security between local applications.

The TP Toolkit architecture is a uniform solution that works with any communication system, both locally and remotely, without compromising security or performance.

The flow of a remote invocation (e.g., RPC) using the TP Toolkit is contrasted with one using a separate transaction manager in Figure 2. The TP Toolkit flow involves only one inter-process message; all others are calls to library procedures linked into the application programs. If the transaction manager insists on intercepting all remote communication, an additional inter-process message is required. The transaction manager must also be a trusted third party, and must preserve the security guarantees between the client and server.

The messages and logging that take place during commit processing are compared in Figure 3. The commit coordination function is part of the TRAN library in one of the participants in the TP Toolkit case. The traditional system with an external transaction manager requires additional messages to this third party, and an additional log record for it to record its state.

Conclusions

The availability of open operating system interfaces and basic distributed systems tools have opened the door for distributed transaction processing on affordable, modern hardware. Transaction processing systems need to offer highly modular, open interfaces to complete the transition. The Transarc TP Toolkit has achieved an unprecedented separation of function in the architecture of low-level transaction processing software.

The TP Toolkit transaction management module provides a rich, high-function computing model and exposes well-defined interfaces to other components without incurring extra cost. The advanced functions permit interoperability with other transaction system models and standards, such as an APPC interface to the LU6.2 protocol and the X/Open XA programming interface. The communication interface allows the use of existing and emerging standard protocols, such as ISO TP and NCS. The recovery interface can be integrated into existing servers without changing their locking, logging, or permanent storage modules. Finally, the architecture minimizes the number of messages and forced log writes without compromising uniformity, modularity, portability, or security.

References

- [Gray78] James N. Gray, "Notes on Database Operating Systems", in *Operating Systems - An Advanced Course*, pages 393-481, Springer-Verlag, 1978.
- [Moss85] J. Eliot B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, MIT Press, 1985.
- [IBM84] IBM, *Guide to IMS/VS V1 R3 Data Entry Data Base (DEDB) Facility*, Document Number GG24-1633, May 1984.
- [Liskov87] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, and W. Weihl, *Argus Reference Manual*, MIT Laboratory for Computer Science, TR-400, November, 1987.
- [Eppin91] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, *Camelot and Avalon: A Distributed Transaction Facility*, Morgan Kaufmann Publishers, 1991.
- [IBM85] IBM, *Systems Network Architecture. Format and Protocols Reference Manual: Architecture Logic for LU Type 6.2*, Document Number SC30-3269-3, 1985.
- [ISO90] International Organization for Standardization, *Information technology -- Open Systems Interconnection -- Distributed transaction processing*, Draft International Standard ISO/IEC DIS 10026-1, 1990.
- [XOPEN90] X/Open, *Distributed Transaction Processing: The XA Specification*, Preliminary Specification XO/PRELIM/90/020, April 1990.

Michael Young is a senior systems designer at the Transarc Corporation. He enjoys working on all sorts of systems software. Now, he's working on basic transaction management. Previously, he was one of the original Mach operating system designers, and for his work on its external paging facility, he was granted a Ph.D. from Carnegie Mellon. To contact him, send electronic mail to "mwyoung+@transarc.com" or U.S. mail to the Transarc Corporation, 707 Grant Street, Pittsburgh, PA 15219.



Dean S. Thompson is the manager of software engineering for Transarc Corporation's transaction processing products. Before that he implemented the recovery manager for the Camelot experimental transaction system at Carnegie Mellon. Dean received his B.S. in Mathematics from Carnegie Mellon in 1989. He reads electronic mail as dt@transarc.com. He reads U.S. mail at Transarc Corporation, 707 Grant Street; Pittsburgh, PA 15219.



Elliot Jaffe is a systems designer at Transarc Corp. His interests include transaction protocols, software engineering and performance optimization. He is currently doing technical product support. Previously he worked on the Camelot system at Carnegie Mellon. He received his B.S. in Mathematics from Carnegie Mellon in 1985. Elliot reads electronic mail at jaffe@transarc.com. He reads U.S. mail at Transarc Corp., 707 Grant Street; Pittsburgh, PA 15219.



The USENIX Association

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *login*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
Telephone: 415-528-8649
Email: office@usenix.org
Fax: 415-548-5738

USENIX Supporting Members

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology Corporation
mt Xinu
Matsushita Graphic Communication Systems, Inc.
Open Software Foundation
Quality Micro Systems
Sun Microsystems, Inc.
Sybase, Inc.

